UNIVERSITE DE LAUSANNE
FACULTE DES HAUTES ETUDES COMMERCIALES

ON SOLVING FAIR EXCHANGE AND RELATED DISTRIBUTED PROBLEMS IN BYZANTINE ENVIRONMENTS

THESE

Présentée à la Faculté des HEC
de l'Université de Lausanne

par

Ian RICKEBUSCH

Licencié en Economie Politique
de l'Université de Lausanne

Titulaire d'un diplôme postgrade en Informatique et Organisation
de l'Université de Lausanne

Pour l'obtention du grade de
Docteur en Systèmes d'Information

2007

UNIL | Université de Lausanne
HEC Lausanne
Le Doyen
Bâtiment Internef
CH-1015 Lausanne

IMPRIMATUR

Sans se prononcer sur les opinions de l'auteur, le Conseil de la Faculté des hautes études commerciales de l'Université de Lausanne autorise l'impression de la thèse de Monsieur Ian RICKEBUSCH, licencié en économie politique de l'Université de Lausanne, titulaire d'un master en informatique et organisation de l'Université de Lausanne, en vue de l'obtention du grade de docteur en Systèmes d'Information.

La thèse est intitulée :

ON SOLVING FAIR EXCHANGE AND RELATED DISTRIBUTED PROBLEMS IN BYZANTINE ENVIRONMENTS

Lausanne, le 5 juin 2007

Le doyen

Suzanne de Tréville

HEC LAUSANNE

Tél. ++41 21 692 33 66 | Fax ++41 21 692 33 65 | www.unil.ch/hec

Jury de Thèse

Monsieur Benoît Garbinato
Professeur à la Faculté des Hautes Etudes Commerciales de l'Université de Lausanne. Directeur de thèse.

Monsieur Yves Pigneur
Professeur à la Faculté des Hautes Etudes Commerciales de l'Université de Lausanne. Expert interne.

Monsieur Marco Tomassini
Professeur à la Faculté des Hautes Etudes Commerciales de l'Université de Lausanne. Expert interne.

Monsieur Pascal Felber
Professeur à la Faculté des Sciences de l'Université de Neuchâtel. Expert externe.

Monsieur François Pacull
Chercheur au Centre européen de recherche Xerox à Grenoble. Expert externe.

Monsieur Luís Rodrigues
Professeur à la Faculté des Sciences de l'Université de Lisbonne. Expert externe.

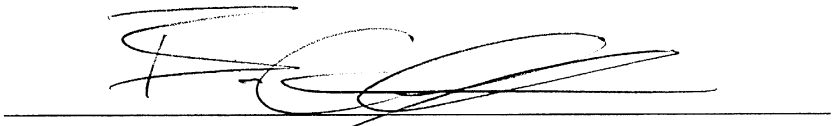
Université de Lausanne
Faculté des HEC

Doctorat en Systèmes d'Information

Par la présente, je certifie avoir examiné la thèse de doctorat de

Ian Rickebusch

Sa thèse remplit les exigences liées à un travail de doctorat.
Toutes les révisions que les membres du jury et le soussigné ont demandées durant le colloque de thèse ont été prises en considération et reçoivent ici mon approbation.



Prof. Benoît Garbinato
Directeur de thèse

Université de Lausanne
Faculté des HEC

Doctorat en Systèmes d'Information

Par la présente, je certifie avoir examiné la thèse de doctorat de

Ian Rickebusch

Sa thèse remplit les exigences liées à un travail de doctorat.
Toutes les révisions que les membres du jury et le soussigné ont demandées durant le colloque de thèse ont été prises en considération et reçoivent ici mon approbation

Yves Pigneur

Prof. Yves Pigneur
Membre interne du jury

Université de Lausanne
Faculté des HEC

Doctorat en Systèmes d'Information

Par la présente, je certifie avoir examiné la thèse de doctorat de

Ian Rickebusch

Sa thèse remplit les exigences liées à un travail de doctorat.
Toutes les révisions que les membres du jury et le soussigné ont demandées durant le colloque de thèse ont été prises en considération et reçoivent ici mon approbation

Prof. Marco Tomassini
Membre interne du jury

Université de Lausanne
Faculté des HEC

Doctorat en Systèmes d'Information

Par la présente, je certifie avoir examiné la thèse de doctorat de

Ian Rickebusch

Sa thèse remplit les exigences liées à un travail de doctorat.
Toutes les révisions que les membres du jury et le soussigné ont demandées durant le colloque de thèse ont été prises en considération et reçoivent ici mon approbation

P. Felber

Prof. Pascal Felber
Membre externe du jury

Université de Lausanne
Faculté des HEC

Doctorat en Systèmes d'Information

Par la présente, je certifie avoir examiné la thèse de doctorat de

Ian Rickebusch

Sa thèse remplit les exigences liées à un travail de doctorat.
Toutes les révisions que les membres du jury et le soussigné ont demandées durant le colloque de thèse ont été prises en considération et reçoivent ici mon approbation

Dr. François Pacull
Membre externe du jury

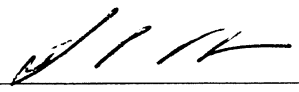
Université de Lausanne
Faculté des HEC

Doctorat en Systèmes d'Information

Par la présente, je certifie avoir examiné la thèse de doctorat de

Ian Rickebusch

Sa thèse remplit les exigences liées à un travail de doctorat.
Toutes les révisions que les membres du jury et le soussigné ont demandées durant le colloque de thèse ont été prises en considération et reçoivent ici mon approbation



Prof. Luís Rodrigues
Membre externe du jury

Abstract

The solvability of the problem of *fair exchange* in a synchronous system subject to Byzantine failures is investigated in this work. The fair exchange problem arises when a group of processes are required to exchange digital items in a fair manner, which means that either each process obtains the item it was expecting or no process obtains any information on the inputs of others.

After introducing a novel specification of fair exchange that clearly separates safety and liveness, we give an overview of the difficulty of solving such a problem in the context of a fully-connected topology. On one hand, we show that no solution to fair exchange exists in the absence of an identified process that every process can trust a priori; on the other, a well-known solution to fair exchange relying on a *trusted third party* is recalled. These two results lead us to complete our system model with a flexible representation of the notion of trust. We then show that fair exchange is solvable if and only if a connectivity condition, named the *reachable majority* condition, is satisfied. The necessity of the condition is proven by an impossibility result and its sufficiency by presenting a general solution to fair exchange relying on a set of trusted processes.

The focus is then turned towards a specific network topology in order to provide a fully decentralized, yet realistic, solution to fair exchange. The general solution mentioned above is optimized by reducing the computational load assumed by trusted processes as far as possible. Accordingly, our fair exchange protocol relies on trusted tamperproof modules that have limited communication abilities and are only required in key steps of the algorithm. This modular solution is then implemented in the context of a pedagogical application developed for illustrating and apprehending the complexity of fair exchange. This application, which also includes the implementation of a wide range of Byzantine behaviors, allows executions of the algorithm to be set up and monitored through a graphical display.

Surprisingly, some of our results on fair exchange seem contradictory with those found in the literature of *secure multiparty computation*, a problem from the field of modern cryptography, although the two problems have much in

common. Both problems are closely related to the notion of trusted third party, but their approaches and descriptions differ greatly. By introducing a common specification framework, a comparison is proposed in order to clarify their differences and the possible origins of the confusion between them. This leads us to introduce the problem of *generalized fair computation*, a generalization of fair exchange. Finally, a solution to this new problem is given by generalizing our modular solution to fair exchange.

Acknowledgements

First and foremost, I wish to express my profound gratitude to Professor Benoît Garbinato. He chose to believe in me at a time when I did not, and he guided me with great benevolence and sensitivity through my doubts and difficulties. I have the utmost respect for him as an advisor, a researcher, a teacher and, above all, as a person.

I wish to thank the members of my jury, Professors Pascal Felber, Yves Pigneur, Luís Rodrigues, Marco Tomassini and Doctor François Pacull, for the time spent examining this thesis, the precious inputs provided and the general interest shown for my work.

I would like to further express my gratitude to Professor Yves Pigneur, the director of the Information Systems Institute, for all the work and energy he has put into managing the institute during all these years, and for always caring about the wellbeing of others before his own. He is the true soul of our institute.

I am grateful to Professor Pierre Bonzon for sharing his knowledge with passion, which has greatly contributed to developing my taste for programming and computer science in general.

I would like to thank sincerely Ivan, Adrian and Mona, my colleagues from office 128.2, for creating such a great atmosphere during all these years and for putting up with a lot of things, not to mention salsa. It was a real pleasure to have them as colleagues and now as friends.

I am thankful to Marco (Lalos), for perpetuating the friendship between the 5th and 1st floor that was initiated by our respective predecessors, and for our *mens-sana-in-corpore-sano* running sessions.

I wish to thank all the people who have shared a break or two (per day) with me over the years, i.e., Jan, Giovanni, Dominique, Alexander, Philippe R., Manu, Sami, Antoine, Mathias, Yelena, Flavio, Laure, Marika, Sabine, Jean-Sé, Bruno, Philippe J., Raul, Diego, Samuel, Amandine, Marc, Stéphane, Sébastien, Daniel, Michael, Katharina, Nils, Samyr, Rym, Frédéric, Grégoire, Florian, to name a few.

I am thankful to François and Vedat for all the work done on the implementation of the modular solution of fair exchange.

I also want to thank all the people working for or around HEC Lausanne, the colleagues from the Information Systems Institute and from the Inforge, the Infocentre team, Conceição and the ladies from the cafeteria

I would like to express many thanks to Sylvain, Vanessa, Jean-Pierre, Patrick, José, Miguel, Ana Maria, Jasmine, Linda, Cristian and all my friends, the old ones, the new ones, those who live nearby, those who live far away, the salseros and those who don't dance, for their faithful support and all the great moments we shared.

Finally, I am very grateful to my mother, my father, my sister, my tonton Yves, Yvette, Daniela, Guilherme, Elizabete and my entire family for their unconditional support, and I wish to further thank my mother, who, on top of all her love and caring, has thoroughly corrected the English of my thesis.

Contents

Introduction **1**

Context 1

Motivation 3

Contribution 4

Structure 7

I Problem & Solvability 9

1 Fair Exchange 11

1.1 Introduction 11

1.2 System Model 13

1.2.1 Topology and Synchrony. 13

1.2.2 Executions and Failure Patterns 14

1.3 A Formal Specification of Fair Exchange 14

1.3.1 Fair Exchange as a Service 15

1.3.2 Fair Exchange Properties 16

1.4 Limitation of the Model 17

1.4.1 Impossibility Result 18

1.4.2 The Trusted Third Party (TTP) 19

1.4.3 How much trust is really needed? 20

2 Trust and Solvability of Fair Exchange 23

2.1 Introduction 23
2.2 System Model: Adding Trust 25
2.2.1 Topology and Synchrony. 25
2.2.2 Executions and Failure Patterns 26
2.3 Solvability in the Model with Trust 27
2.3.1 The Reachable Majority (RM) Condition 28
2.3.2 Impossibility Result 29
2.3.3 Solvability Result 31
2.4 Fair Exchange under the RM Condition 31
2.4.1 Best-effort Multicast 32
2.4.2 Byzantine Agreement 33
2.4.3 A General Algorithm of Fair Exchange 34
2.4.4 Examples of Executions 36
2.4.5 Correctness Proof 38
2.4.6 Best-effort Multicast: Solution and Proof 40
2.4.7 Implementation of BA in our Model 42
2.5 Revisiting Existing Solutions 44
2.5.1 The Trusted Third Party 44
2.5.2 The Guardian Angels 45

II Solution & Implementation 47

3 A Modular Solution 49

3.1 Introduction 49
3.2 System Model: a Specific Topology 51
3.2.1 Topology and Synchrony. 52
3.2.2 Executions and Failure Patterns 52
3.3 Fair Exchange with Secure Boxes 53
3.3.1 Secure Box 53

3.3.2 Byzantine Agreement 54
3.3.3 A Specific Algorithm of Fair Exchange 55
3.3.4 Examples of Executions 57
3.3.5 Correctness Proof 57
3.4 Discussion 61
3.4.1 Are we being unfair to Byzantine processes? 61
3.4.2 Complexity Analysis 62

4 A Java Implementation 65
4.1 Introduction 65
4.1.1 From a Pseudo to a Real Programming Language 66
4.2 A Java Implementation of Fair Exchange 68
4.2.1 Static Model 68
4.2.2 Dynamic Model 69
4.2.3 Byzantine Agreement Module 72
4.2.4 Secure Box Module 73
4.2.5 Network Module 74
4.3 Implementing Byzantine Behaviors 75
4.3.1 Refactoring the Correct Implementation 75
4.3.2 From Correct to Byzantine through Inheritance 77
4.4 A Visualization Tool 79
4.4.1 Execution Setup 80
4.4.2 Running an Execution 81
4.4.3 Execution Monitor 82

III Comparison & Generalization 83

5 Fair Exchange vs. Secure Multiparty Computation 85
5.1 Introduction 85
5.1.1 Secure Multiparty Computation (SMC): a Brief History . 87

5.2 An Integrated Specification Framework 88
5.2.1 Functional Definition 89
5.2.2 Behavioral Constraints 90
5.3 Revisiting the Specification of the SMC Problem 91
5.3.1 Functional Definition 91
5.3.2 Behavioral Constraints 91
5.4 Revisiting the Specification of the Fair Exchange (FE) Problem 92
5.4.1 Functional Definition 92
5.4.2 Behavioral Constraints 93
5.5 Comparing SMC and FE 94
5.5.1 The Respective Strengths of SMC and FE 94
5.5.2 Towards Generalized Fair Computation 95
5.6 Origin of the Confusion 96
5.6.1 Contradictory Results? 96
5.6.2 An Ideal Model? 97

6 Generalized Fair Computation 99
6.1 Introduction 99
6.1.1 Fair Computation, no Support for Fair Exchange 101
6.2 The Generalized Fair Computation (GFC) Problem 102
6.2.1 Functional Definition 102
6.2.2 Behavioral Constraints 103
6.3 Using GFC for Solving SMC and FE 104
6.3.1 An Algorithm for SMC based on GFC 104
6.3.2 Correctness Proof 105
6.4 Solving GFC in the Absence of a TTP 105
6.4.1 Relying on Specific Modules 106
6.4.2 An Algorithm Solving GFC 108
6.4.3 Correctness Proof 110

Conclusion **113**

Research Assessment 113

Future Research 115

Final Word 116

Bibliography **116**

Figures

An example of a distributed system. 1
Thesis outline. 6

1.1 Thesis outline. 12
1.2 A fully connected topology, with five processes. 13
1.3 A topology relying on a centralized *trusted third party* (TTP). . 20
1.4 How much trust is needed for solving fair exchange? 21

2.1 Thesis outline. 24
2.2 Examples of valid topologies as defined in our model. 26
2.3 How much trust is needed for solving fair exchange? 27
2.4 Topologies under the RM condition. 28
2.5 Layered diagrams of the modules involved in our solution. . . . 32
2.6 Examples of executions of Algorithms 2.1 and 2.2. 37
2.7 Three steps for adapting our context to the original setting of Byzantine agreement. 43
2.8 The TTP topology allows any number of Byzantine processes. . 44
2.9 The Guardian Angels topology requires an honest majority. . . . 45

3.1 Thesis outline. 50
3.2 Topology with five participants and their trustees. 52
3.3 A layered diagram of the modules involved in our solution. . . . 53
3.4 Successful executions of Algorithm 3.1. 58
3.5 Aborted executions of Algorithm 3.1. 59

4.1 Thesis outline. 66
4.2 Communication architecture. 67
4.3 Class diagram. 68
4.4 Sequence diagram. 70
4.5 Threads running Algorithm 3.1. 71
4.6 Class diagram of the Byzantine behaviors. 77
4.7 Extended class diagram. 79
4.8 Setup window. 80
4.9 Execution sequence of the graphic user interface. 81
4.10 Monitoring window. 82

5.1 Thesis outline. 86
5.2 A fully connected topology, with five processes. 89
5.3 Illustration of the generality and security levels of the problems and their relations. 95

6.1 Thesis outline. 100
6.2 The generality and security levels of the various problems. . . . 101
6.3 Topology with five participants and their trustees. 106
6.4 A layered diagram of the modules involved in Algorithm 6.2. . . 107

Tables

4.1 Number of messages per process in a given round of BA. 74

5.1 Specifications in modern cryptography and distributed systems. 90

Algorithms

2.1 Fair exchange – Protocol executed by participant p_i 35
2.2 Fair exchange – Protocol executed by trustee p'_i 36
2.3 Best-effort multicast protocol executed by process p_i 41
3.1 Fair exchange protocol executed by process p_i 56
6.1 SMC protocol executed by process p_i 104
6.2 Generalized fair computation protocol executed by p_i 109

Code Listings

4.1 Java method before modification. 75
4.2 Java method after modification. 76
4.3 Implementing a method to allow crash-recovery behaviors. . . . 76
4.4 Overriding a method to produce a crash-stop behavior. 77
4.5 Run method for the crash-recovery behavior. 78

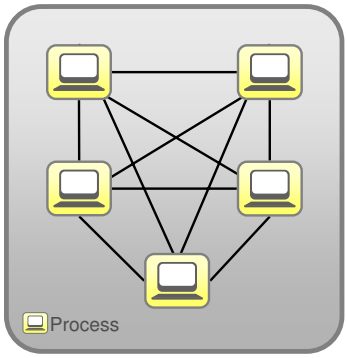
Introduction

A child of five would understand this.
Send someone to fetch a child of five.

Groucho Marx

Context

In real life, distributed systems are composed of many and varied elements of infrastructure. For example, computers do not all provide the same computational power or storage capacity. Studying such systems from a general standpoint must therefore rely on basic abstractions to capture the essence of distributed systems [GR06d].



An example of a distributed system.

Modeling Distributed Systems

A distributed system consists of a set of computers linked by communication channels, so the two primary abstractions are respectively *processes* and *links*, as illustrated in the figure above. In order to represent real conditions, both these abstractions are characterized by failure models, i.e., descriptions of the possible failures that may be exhibited. Instances of the *process* abstraction are distinguished by their behaviors.

- *Crash-stop failures* — In this form of failure, processes are either correct or crashed. However once they have crashed, they do not go back to being correct.
- *Crash-recovery failures* — This definition of failure allows for recovering from a crash. A *good* process is one that either never crashes or eventually stops crashing. All others are said to be *bad* [ACT98].
- *Byzantine failures*[1] — With Byzantine failures, processes may exhibit any kind of behavior. This includes crash-stop and crash-recovery failures and more specifically *malicious* ones, i.e., resulting from the purposeful design of an adversary.

Similarly, instances of the *link* abstraction may vary with respect to their levels of reliability, since messages can be lost during transit through the network.

- *Fair-loss* — This type of link ensures that messages are not systematically dropped during the transmission. However there is no guarantee that a given message will reach its destination.
- *Perfect* — With perfect links, messages are reliably delivered, i.e., the message is eventually received if both the sender and the receiver are correct. Moreover, messages are delivered only once and no message is delivered if it was not previously sent by some process.

A third basic dimension characterizing distributed systems concerns timing assumptions, i.e., process speeds and communication delays.

- *Asynchronous* — This definition indicates that there are no timing assumptions whatsoever made about the system.

[1] Also sometimes called *arbitrary* failures. The term *Byzantine* comes from an analogy introduced in [LSP82] to discuss the problem of reaching *agreement* among processes that may lie to each other.

- *Synchronous* — In the case of a synchronous system, there are upper bounds on the processing and transmission delays, and each process has a local clock which only deviates from others at a bounded rate.

These failure models provide an overview of those commonly found in the literature [GR06d], nevertheless, none of the lists is exhaustive. While various combinations of the three categories are valid system models, in this thesis we consider a *synchronous* distributed system with *Byzantine* failures and *perfect* links.

Common Problems in Distributed Systems

Independently of the definition of the system model, a wide range of problems are studied in the field of distributed systems. The problems of *consensus* and *reliable broadcast* are arguably the cornerstones of the domain. While the former allows processes to agree on a common value belonging to the set of values that each process has proposed, the latter provides a means of reliably sending a message to a group of processes. Both these problems have been heavily studied and come in many variants, such as *best-effort broadcast*, *causal broadcast*, *regular consensus* and *uniform consensus* to name a few [GR06d].

A particular instance of consensus is the problem of *non-blocking atomic commit* (NBAC) [GHM+99], in which, according to how things went, each process votes either *YES*, to commit some local computations, or *NO*, to abort them. Moreover, besides achieving *termination* for all good processes, an important property of NBAC is that it is biased towards abort, i.e., in order to commit all processes must have voted *YES*.

In this thesis, we propose to study the problem of *fair exchange*, which is closely related to the problem of *non-blocking atomic commit* [AGG+04]. Fair exchange consists in allowing processes to exchange a digital item in a fair manner. Terms of the exchange are known a priori, so processes offer an item and expect one in exchange. The outcome of the exchange is fair if either all the processes obtain the expected item or no process receives anything.

Motivation

In the modern world, the notions of fair exchange and trust are ubiquitous: every day, without even noticing, we participate in numerous commercial exchanges, which we expect to be fair (and most actually are). Fundamental to these exchanges is the notion of *trust*. In the physical world, this trust is supported by the identification and the implicit reputation of tangible exchange

partners. In the digital world, on the contrary, fair exchange is a surprisingly difficult problem. This can be explained by the lack of trust that characterizes the digital realm. Yet, fair exchange is a fundamental problem that has constantly been studied over the past decades and that has recently regained interest [AGG+04, AGGV05, AV03]. This is partly due to the advent of *m*-business[2] as a natural evolution of *e*-business, i.e., extending the possibilities of *e*-business through the use of mobile devices, e.g., cellular phones. When it comes to solving fair exchange in such semi-open environments, i.e., where all parties are not necessarily identified a priori, carefully modeling and analyzing trust relationships between processes is a key issue.

Most successful *e*-business solutions today follow a classical (centralized) client-server architecture. This implies that current *e*-business solutions somehow fail to take full advantage of the Internet's underlying protocols, which were designed to support fully decentralized approaches.[3] For example, current *e*-business solutions do not provide a favorable environment for electronic exchanges in the absence of some centralized and trusted server, i.e., they fail to support peer-to-peer only settings. On the other hand, the emergence of mobile devices and ad hoc networks, which often have to operate in a disconnected manner with respect to the Internet, is forcing us to reconsider the current *e*-business architectures. In that respect, we maintain that fair exchange is a keystone for peer-to-peer *m*-business interactions at the middleware level. By studying the solvability of fair exchange in various contexts, and more specifically in decentralized ones, this thesis proposes fair exchange as a basic building-block and as a first step towards achieving peer-to-peer *m*-business solutions.

Contribution

This thesis discusses the solvability of the problem of fair exchange and some related aspects. Its contribution is threefold, respectively divided into the three parts of the structure illustrated on page 6, and mostly based on three published companion papers [GR06a, GR06b, GR06c] and a technical report, which is currently under review in an international conference at the time of writing [GR07].

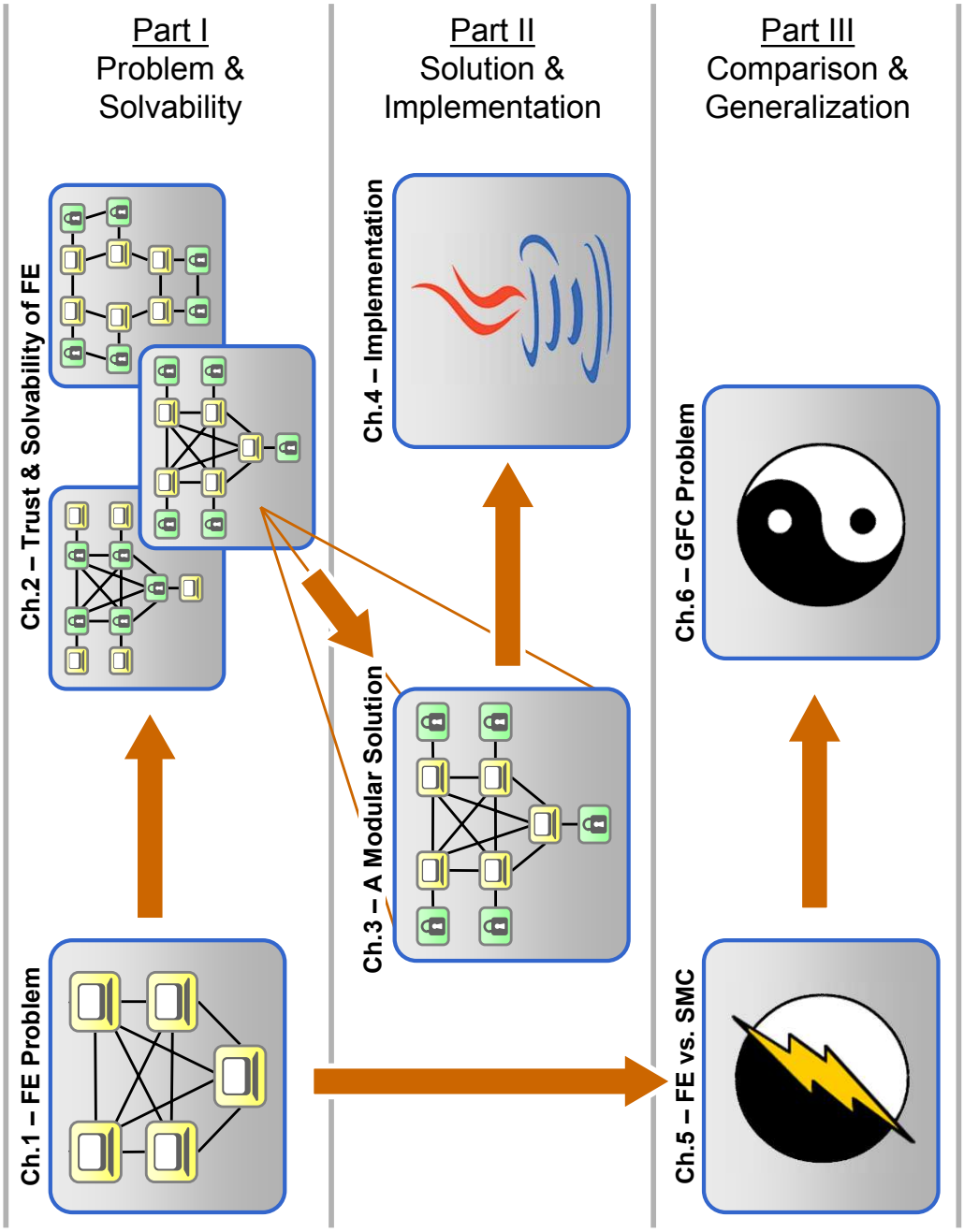
[2] *Mobile*-business, i.e., business pursued by relying on the infrastructure provided by mobile technology, such as mobile phones.

[3] The ARPANET project aimed at building a network with no single point of failure, in order to survive a nuclear strike.

Part I: Problem & Solvability. The first contribution of the thesis is to present a study of the solvability of fair exchange in a general context and its links with the notion of trust. In doing this, we introduce a novel specification of the problem of fair exchange. By first assuming a general system model where no process can be trusted, we show that there is no solution to the problem of fair exchange. Two documented solutions both relying on some sort of trusted entity – the *trusted third party* (TTP) [BP90] and the *guardian angels* [AV03] – lead us to propose an extension of the general model to include a flexible representation of the notion of trust. In this new model, we show a connectivity condition, both necessary and sufficient, for fair exchange, named the *reachable majority* condition. Sufficiency is proved by presenting a general solution achieving fair exchange under the *reachable majority* condition.

Part II: Solution & Implementation. While the reachable majority condition requires a minimum level of trust in order for fair exchanges to occur, our second contribution is to provide a fully-decentralized, yet realistic, solution to fair exchange. As in the *guardian angels* solution [AV03], the context is that of a network topology of fully connected processes, each hosting a tamperproof module that can therefore be trusted. However, in order to propose a solution as realistic as possible, the embedded tamperproof modules are only used in limited key steps of the protocol. This solution is then implemented in the context of a pedagogical visualization tool, in which executions of our fair exchange protocol can be configured with various settings, such as the behaviors of each process, and monitored through a graphical display.

Part III: Comparison & Generalization. Our third contribution is to propose a comparison of fair exchange and the problem of *secure multiparty computation*. Indeed, these two problems are apparently similar but certain results found in their respective literature are confusingly contradictory. The wide differences of description and approach in their respective research fields render a straightforward comparison hazardous. Our first step is thus to introduce a common specification framework and, using this, revisit the descriptions of both problems. The gap between fair exchange and secure multiparty computation then leads us to describe a third problem, i.e., *generalized fair computation*, which is a generalization of the problem of fair exchange. We thus present a generalization of our solution to fair exchange that solves the newly introduced problem.



Thesis outline.

Structure

The thesis is divided into six chapters, two in each part, as illustrated on page 6. This representation of the structure is recalled at the beginning of each chapter. As there is no chapter specifically dedicated to the related work, each chapter starts with an introductory section which discusses research work that is close to ours, while setting the context and linking the current chapter to previous ones.

The figure of page 6 also illustrates the logical connections between the chapters. As shown, the first three chapters link together linearly since, in these, theoretical aspects of the problem of fair exchange and its solvability are discussed sequentially. Chapter 4 is linked to Chapter 3 since it offers a concrete Java implementation of the solution of that chapter. However, Chapter 4 stands somewhat apart from the others, as it presents aspects related to fair exchange that are more practical. Since the implementation does not come with performance results, the research contributions of the chapter are weaker. The reader may therefore choose to postpone reading it and continue with Chapters 5 and 6, which discuss theoretical aspects logically linked to those of Chapter 1. By comparing definition and specification aspects of fair exchange and secure multiparty computation, the last two chapters address the confusing similarity of the two problems.

While these last chapters, which deal with related work, could have come sooner in the thesis, this structure allows us to focus directly on the solvability of fair exchange. Moreover, as comparison with secure multiparty computation started when our first results on the solvability of fair exchange were challenged by reviewers from the field of modern cryptography, it also corresponds to the chronology of our research.

Thesis Outline

Chapter 1 introduces a novel specification of the problem of fair exchange relying on a set of elemental properties and presents a first impossibility result stating that fair exchange is impossible in the absence of trust.

Chapter 2 extends the system model by adding flexible elements capturing the notion of trust. In this new model, it shows that a necessary and sufficient condition for solving fair exchange is a connectivity condition implying that correct processes have reliable access to a majority of the elements of trust. A general solution to the problem of fair exchange is presented as part of the proof of the sufficiency of the condition.

Chapter 3 focuses on a specific topology of the extended model to propose a

realistic decentralized solution to fair exchange. By relying on tamperproof secure modules for only key steps of the protocol but nonetheless providing the required trust, our modular solution achieves fair exchange in the context of an honest majority.

By providing both an implementation of the modular solution presented in Chapter 3 and a graphical user interface, Chapter 4 presents a pedagogical visualization tool for monitoring specific executions of fair exchange. The application provides an interesting instrument for apprehending the difficulty of fair exchange. Chapter 4 also presents a general approach to implementing a wide range of Byzantine behaviors, as is indeed necessary for illustration purposes. From a research contribution perspective, this chapter is somewhat weaker than the others as it does not for example provide performance results.

Chapter 5 proposes a comparison of fair exchange and secure multiparty computation, a similar problem coming from the research fields of modern cryptography. After a brief presentation of the problem of secure multiparty computation, highlighting the heterogeneity of approach and specification of the two problems, a common specification framework is introduced. The problems are then revisited using the new framework, allowing a more accurate comparison.

Directly deriving from the comparison of Chapter 5, Chapter 6 proposes a generalization of fair exchange, named generalized fair computation. Based on the solution of Chapter 3 with secure boxes, a general solution to the new problem is then presented.

Part I

Problem & Solvability

Chapter 1

Fair Exchange

The devil hides in details.

Swiss Proverb

Abstract. This chapter presents the problem of *fair exchange*, and gives a first overview of the difficulty of solving such a problem in the context of a fully-connected synchronous distributed system. Our first contribution is a novel specification of the fair exchange problem that clearly separates safety and liveness. In the context of a synchronous model where processes communicate by message passing and might behave maliciously, it is shown that no solution to fair exchange exists in the absence of an identified process that every process can trust a priori. This trust-related impossibility result constitutes the second contribution of this chapter. Finally, we discuss a well-known solution to the problem of fair exchange relying on a *trusted third party*.

1.1 Introduction

Various specifications of the fair exchange problem have been proposed, with sets of properties differing slightly [ASW00, Ate99, AGGV05, FR97, Mic03, PG99, RRN05], particularly in the notion of fairness, which is the most difficult to capture [MGK02, RRN05]. Most specifications are actually meaningful for exchanges involving only two processes, i.e., they are impossible to satisfy in models allowing more than one Byzantine process. Our specification of fair exchange, on the contrary, considers the general case where more than

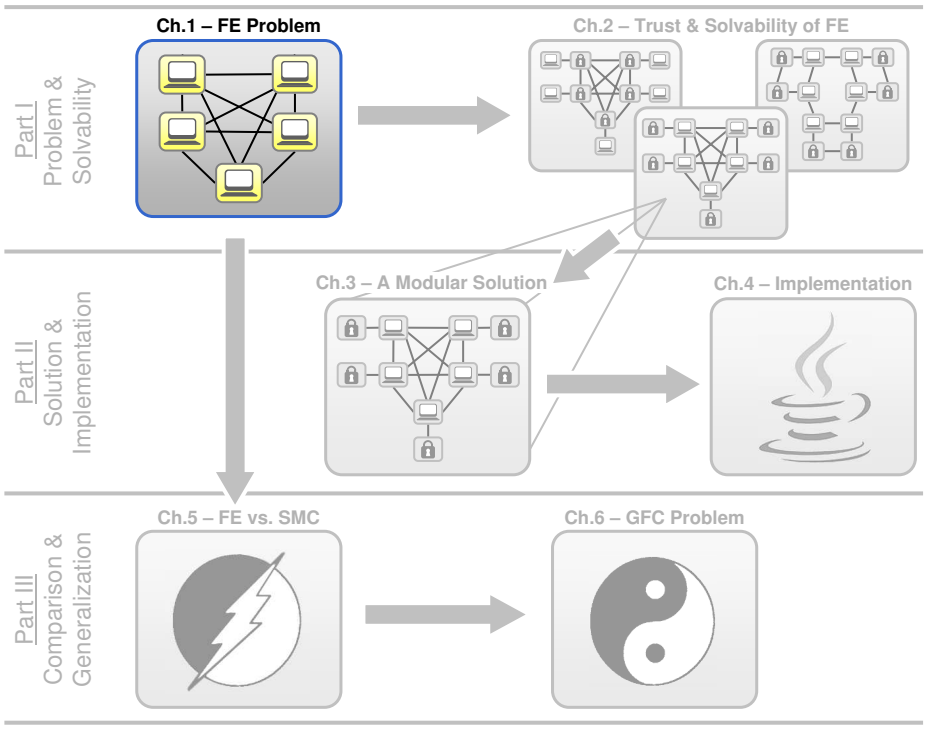


Figure 1.1: Thesis outline.

two processes might be involved. Moreover, we propose a specification relying on six properties clearly distinguished by liveness or safety.

Besides proposing a specification or a solution, some authors also discuss the difficulty of fair exchange and propose impossibility results in various models. In [PG99], fair exchange is measured against consensus, and an impossibility result on fair exchange in asynchronous models is shown by comparison with the FLP impossibility [FLP85]. In [EY80], fair exchange is shown to be impossible to solve *deterministically* in an asynchronous system with no *trusted third party* (TTP). In this chapter, we show that fair exchange cannot be solved in a synchronous model in the absence of some identified process trusted *a priori* by every other process.

This chapter thus provides the context and groundwork for our study of fair exchange and its link with trust, as recalled in Figure 1.1. To keep the field of application as wide as possible, the context is set in a general synchronous system model, i.e., with no specific topology or failure assumptions. It is in this wide context that we propose our novel specification of fair exchange.

1.2 System Model

We consider a general distributed system consisting of a set Π of n processes, $\Pi = \{p_1, \ldots, p_n\}$. As detailed hereafter, processes form a connected graph and may exhibit Byzantine behaviors. Figure 1.2 illustrates a possible setting, with five fully connected processes.

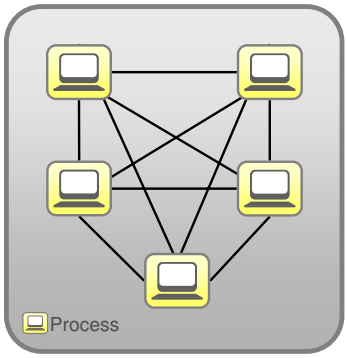


Figure 1.2: A fully connected topology, with five processes.

1.2.1 Topology and Synchrony.

Processes are interconnected by some communication network and communicate by message passing. The system is *synchronous*: it exhibits *synchronous computation* and *synchronous communication*, i.e., there exist upper bounds on processing and communication delays. We also assume the existence of some global real time clock, whose tick range, noted T, is the set of natural numbers. The global clock is virtual in the sense that processes do not have access to it.

Regarding the network topology, we merely assume that processes of Π form a connected graph. Links are reliable bidirectional communication channels, i.e., if both sender and receiver are correct, any message inserted in the channel is *eventually* delivered by the receiver.[1] Formally, such channels are said to be *perfect links* (PL), which provide *send* and *deliver* primitives (respectively PL.send() and PL.deliver() functions) and ensure the following set of properties [GR06d].

Reliable delivery. Let p_i be any process that sends a message m to a process p_j. If neither p_i nor p_j crashes, then p_j eventually delivers m.

[1] This somewhat counterintuitive use of the verb *deliver* is commonly used in the field of distributed systems and implies that the *deliver* function of the receiver is triggered by the lower layers [GR06d, HT93].

No duplication. No message is delivered by a process more than once.

No creation. If a message m is delivered by some process p_j, then m was previously sent to p_j by some process p_i.

The *synchronous* system assumption implies that the delivery will occur within some known time bound Δ_{PL}.

1.2.2 Executions and Failure Patterns

The *execution* of an algorithm A is defined as a sequence of steps executed by processes of Π. In each step, a process has the opportunity to perform atomically all three following actions: (1) send a message, (2) receive a message and (3) update its local state.[2] Based on this definition, a *Byzantine process* is one that deviates from A in any way, so a Byzantine process is Byzantine against a specific algorithm A. It is a known result that Byzantine failure can only be defined with respect to some algorithm [DGG05]. A *Byzantine failure pattern* f is then defined as a function from T to 2^{Π} where $f(t)$ denotes a set of Byzantine processes that have deviated from A through time t. A failure pattern f can thus be seen as a projection of all process failures during some execution of A. Once a process starts misbehaving, it cannot subsequently be considered correct, i.e., $f(t) \subseteq f(t+1)$. All the above definitions regarding executions and failures are similar to the models of [CT96, DGG05].

An important result regarding Byzantine failures is that they cannot be detected effectively. In [DGG02], it is shown that, because of the inherent nature of Byzantine failures, a failure detector can only be built with respect to the algorithm using it. Contrary to crash-stop failures, Byzantine failures are not independent of the execution of the algorithm. Thus, Byzantine processes can only be detected once they start deviating from the algorithm they are supposed to run. In other words, as long as a Byzantine process behaves according to the algorithm, it cannot be distinguished from a correct process.

1.3 A Formal Specification of Fair Exchange

The fair exchange problem consists in a group of processes trying to exchange digital items in a fair manner. The difficulty of the problem resides in achieving fairness. Intuitively, fairness means that, if one process obtains the expected digital item, then all processes involved in the exchange should also obtain

[2] In each step, the process can of course choose to skip any of these actions, e.g., if it has nothing to send.

their expected digital items. The assumption is made that each process knows both the set Π of processes participating and the terms of the exchange.

The terms of the exchange are defined by a set Ω of pairs of processes (p_i, p_j) and a set D of item descriptions, $D = \{d_1, \ldots, d_n\}$. A description d_i is the description of the item expected by process p_i. Furthermore d_i is unique, so if $i \neq j$, then $d_i \neq d_j$. This restriction eliminates cases where processes are expecting the exact same item. However, such cases can be easily addressed by distinguishing identical items with a special tag, e.g., the ID of the offering process. An element of Ω is a pair (p_i, p_j), where p_j is the receiver of the item offered by p_i. Elements of Ω are defined such that p_j is the image of p_i through a bijective map (or permutation) of Π, with $i \neq j$. Let M denote the set of digital items *actually* offered during an execution of fair exchange, $M = \{m_1, \ldots, m_n\}$, with m_i being offered by process p_i. Accordingly, for each description in D a corresponding item in M does not necessarily exist, since M includes items that might have been offered by Byzantine processes. Finally, let $\mathsf{desc}(m)$ be the function returning the description of item m.

1.3.1 Fair Exchange as a Service

Fair exchange can be seen as a service allowing processes to exchange digital items in a fair manner. Each process offers an item in exchange for a counterpart of which it has the description. The exchange is completed when every process releases the expected counterpart or all processes release the abort item φ, meaning that the exchange has aborted. To achieve this, the service offers the two following primitives.

> **offer**(m_i, p_j) – Enables the process p_i to initiate its participation in the exchange with processes of Π by offering item m_i to p_j, in exchange for the item matching description d_i, with d_i and Π known a priori.
>
> **release**(x) – Works as a callback to inform the process that the exchange is completed. Process p_i receives item x, which is either an item matching d_i or the abort item φ.

At the end of an exchange, we say that p_i *releases* an item, meaning that the service calls back the *release* operation of p_i. This convention is similar to the one used for typical *deliver* primitives, e.g., with reliable broadcast primitives [HT93].

At the heart of our specification lies a key assumption: the *release* primitive is the only way by which any process may obtain its desired item. This implies that no process has access to its desired item until it is safe and fair to release

it. In other words, the item is either physically or virtually out of reach, i.e., it is stored in some unaccessible address space, or it is stored locally but in encrypted form using a key that the local process does not hold.

1.3.2 Fair Exchange Properties

We now specify the formal properties of the fair exchange problem. While several other specifications exist in the literature [AGGV05, AV03, PG99], our specification differs in that it separates safety and liveness via *fine-grained* properties. Such elemental properties then allow us to reason more precisely about the correctness of our solution.

Validity. If a correct process p_i releases an item x, then either $x \in M$ and x matches d_i, or x is the abort item φ. (safety)

Uniqueness. No correct process releases more than once. (safety)

Non-triviality. If all processes are correct, no process releases the abort item φ. (safety)

Termination. Every correct process *eventually* releases an item. (liveness)

Integrity. No process p_j releases an item m_i, with process p_i correct, if m_i matches description d_k of some correct process p_k, with $p_k \neq p_j$. (safety)

Fairness. If any process p_i releases an item m_j matching description d_i, with p_i or p_j correct, then every correct process p_k releases an item matching its description d_k. (safety)

Of these six properties, the last two, *integrity* and *fairness*, are specific to the problem of fair exchange and define precisely the possible outcomes of fair exchange algorithms. The *integrity* property ensures that no process obtains an item offered by a correct process and matching the description of some other correct process. This does not prevent a Byzantine process from illicitly obtaining the item destined for or offered by some other Byzantine process, since such a behavior cannot be prevented and does not prejudice any correct process. The *fairness* property guarantees that if any process obtains its expected item offered by some other process, with at least one of them being correct, then every correct process also obtains its expected item. In other words this property prevents a Byzantine process from taking advantage of a correct process but does not protect other Byzantine processes from their own incorrect behaviors. More trivially, it also ensures that no correct process takes advantage of any process, correct or not.

Note that neither of these properties takes into account the case where a Byzantine process tries to guess the value of one or more items. However, we exclude this *guessing game* case, since the probability of guessing can be made arbitrarily low, as is the case in any cryptography-based guarantees [Gol04].

The definition of *fairness* described above is sometimes characterized by the terms *true*, *perfect* or *deterministic* in order to distinguish it from definitions such as *weak fairness* [RR02] or *probabilistic fairness* [AV03]. The *weak fairness* property does not require the exchange to be fair but rather that correct processes are able to gather evidence of potential misbehaviors, whereas the *probabilistic* one only ensures fairness with a certain probability.

Other specifications of fair exchange usually rely on a single property to capture the notion of *true fairness* [ASW00, AGGV05, PG99]. However, we argue that if those specifications are suitable for cases where $n = 2$, they are impossible to satisfy in models allowing more than one Byzantine process. In [AGGV05], for example, the *fairness* property requires that if any correct process does not obtain its item, then no process obtains any items from any other process. This is clearly unsustainable in the presence of two or more Byzantine processes because one cannot prevent two Byzantine processes from conspiring in order for one of them to obtain the item of the second one. A simple but flawed remedy would be to modify this definition as follows: if any correct process does not obtain its item, then no process obtains any items from any *correct* process. While at first it seems correct, this definition of *fairness* allows a correct process to obtain the item of a Byzantine process, even if other correct processes do not obtain anything. Hence, this specification is still flawed.

1.4 Limitation of the Model

With the context set and the problem defined, there remains the question of solving fair exchange. The difficulty of this resides in ensuring *fairness* to all correct processes. In this section, we show first that ensuring *fairness* is impossible without at least one trusted process, even when assuming a synchronous system and a fully connected topology. We then present a simple solution found in the literature [BP90], which indeed relies on a trusted process, named *trusted third party* (TTP). This process does not take part in the exchange but simply provides the sufficient level of trust for ensuring fairness. We conclude by questioning the necessity of a TTP, i.e., is there some trust setting, weaker than a TTP but sufficient nonetheless for fair exchange?

1.4.1 Impossibility Result

In [EY80], fair exchange is shown to have no solution in an asynchronous model prone to Byzantine failures. In Theorem 1 below, we prove that the fair exchange problem has no deterministic solution, if there is no *trusted process*, even in the context of a perfectly synchronous model. In our model, no such assumption is made about any process of Π, i.e., each process is potentially correct or Byzantine.

Definition 1 (Trusted Process)**.** *A process that is known to be correct a priori by all the other processes, i.e., all processes know that this process will not deviate from its expected behavior.*

For simplicity and without loss of generality, we assume that an item is indivisible, i.e., it cannot be sent in pieces. Allowing items to be broken into pieces, e.g., using techniques from [Sha79], does not ensure *fairness* unless assumptions are made that each process has the same computational power and that receiving only partial data from an item is not entirely useless. If such is not the case, this technique then faces the same fair exchange problem when sending the last piece of item. In any case, since we are concerned with *true fairness*, the indivisible item assumption does not reduce the scope of impossibility. Again for simplicity, we also assume that the item is not ciphered in order to prevent the receiver from having immediate access to it, e.g., by encrypting it with the private key of the sender, since it does not help in any way to solve the problem. Indeed, having to exchange the keys in a fair manner in order for the processes to decipher the items would again produce the same fair exchange problem. Nonetheless, an item may still be encrypted with the public key of the receiver to prevent other processes from intercepting it.

Theorem 1. *In the context of a synchronous model with Byzantine failures, there is no deterministic solution to the fair exchange problem, if there is no trusted process, even in the presence of only a single Byzantine process.*

Proof. The proof is by contradiction.
Assume that some algorithm A solves fair exchange and that there is no trusted process. Consider an execution E of A in which all processes are correct. From the *non-triviality*, *termination* and *validity* properties of fair exchange, in E, every process releases its desired item, and in particular, some process p_i releases item m_j, with m_j matching description d_i and $(p_j, p_i) \in \Omega$, and some process p_k releases item m_i, with m_i matching description d_k and $(p_i, p_k) \in \Omega$. Now, since no process can be trusted and Byzantine processes cannot be detected a priori, in any execution, no process other than p_i and p_k may hold item m_i. Thus we know that in a previous step of E, p_k receives m_i from p_i.

We now consider the two following cases: either (a) p_i sends m_i after receiving m_j or (b) p_i sends m_i before receiving m_j.

Case (a). Since there is no trusted process, if p_i sends m_i after receiving m_j, we can derive an execution E', similar to E, in which p_i is Byzantine and deviates from A after receiving m_j by omitting to send m_i and by releasing m_j. Since no process is trusted, in E', no process other than p_i holds m_i. Thus, from the *no creation* property of perfect links, p_k never receives and thus never releases m_i. To satisfy the *validity* and *termination* properties, in E', p_k releases φ but thus violates *fairness*. Thus, in E, p_i sends m_i before receiving m_j. Furthermore, this is true for every process, so from the definition of Ω and by circular reasoning, in E, all items are sent roughly at the same time. This now leaves us with case (b).

Case (b). We know that, in E, p_i sends m_i before receiving m_j and that all items are sent roughly at the same time. Now, since there is no trusted process, we can derive an execution E'', similar to E, in which p_j is Byzantine and deviates from A by omitting to send m_j. Since all items are sent at the same time, p_j receives and releases some item m_x matching d_j. From the fact that no process can be trusted to hold any item other than its own, in E'', we know that no process other than p_j holds m_j. Thus, from the *no creation* property of perfect links, p_i never receives and thus never releases m_j. To satisfy the *validity* and *termination* properties, in E'', p_i releases φ but thus violates *fairness*. Therefore, algorithm A does not solve fair exchange. A contradiction. □

1.4.2 The Trusted Third Party (TTP)

A simple solution to the problem of fair exchange is to introduce a *trusted third party* (TTP) and most solutions indeed rely on this in some form. A TTP is a trusted process directly accessible to all processes, as shown in Figure 1.3. Fairness is thus trivially ensured by having processes send their items to the TTP, which forwards the items, if the terms of the exchange are fulfilled [BP90]. A TTP brings synchronism and control over terms of the exchange in order to ensure fairness but constitutes a bottleneck and a single point of failure. Moreover, such a centralized solution is unrealistic in the context of a full peer-to-peer system, e.g., as in ad hoc networks.

For this reason, various so-called *optimistic* algorithms have been proposed which only involve the TTP when something goes wrong, i.e., when an attempt to cheat is detected [ASW00, Mic03, BP90, BDM98, BWW00]. However

optimistic approaches are based on the strong assumption that the environment is mostly honest, i.e., most exchanges involve correct processes only. The TTP is thus required for the rare cases where not all processes are correct. To weaken the role of the TTP, in [FR97] for instance, Franklin and Reiter propose a solution using a *semi-trusted* third party that can misbehave on its own but does not conspire with either of the two participant peers. Similarly, the authors of [SXL05] propose a solution based on a cluster of untrusted servers acting as third parties. In the latter paper, however, the authors recognize that they are merely solving a variant of the weak fair exchange.

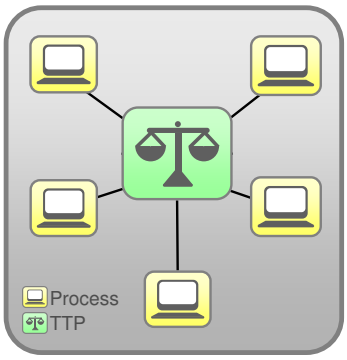


Figure 1.3: A topology relying on a centralized *trusted third party* (TTP).

1.4.3 How much trust is really needed?

We have seen two major results regarding fair exchange, which are illustrated in Figure 1.4. On one hand, our impossibility result shows that in the absence of some trusted process, fair exchange simply cannot be solved. On the other hand, a solution from [BP90] proves that introducing a centralized trusted third party is a strong enough assumption to ensure *fairness*. However, while relying on a TTP is an effective solution, it also has its drawbacks: it implies a centralized architecture, which is inappropriate for peer-to-peer systems, and introduces of a single point of failure.

The question that we thus need to address is whether such a strong centralized solution is really necessary in order to solve fair exchange. In other words, are there not some hidden assumptions embedded in the TTP architecture that are not strictly necessary for solving the problem of fair exchange? These first results provide the groundwork for the next chapter, in which we propose to study the degree of trust, both necessary and sufficient, in order to ensure that solutions to fair exchange exist.

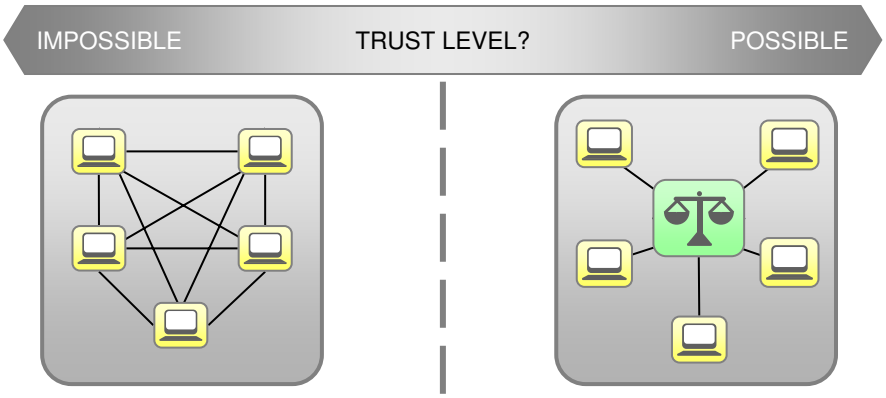


Figure 1.4: How much trust is needed for solving fair exchange?

◊◊◊

Chapter 2

Trust and Solvability of Fair Exchange

To be trusted is a greater compliment than to be loved.

George MacDonald

Abstract. In this chapter, we examine the solvability of fair exchange by introducing a generic model with trust, i.e., with trusted and untrusted processes. We then show that the solvability of fair exchange depends on a necessary and sufficient topological condition, which we name the *reachable majority* condition. The first part of this result, i.e., necessity, is captured by an impossibility result in the context of our model with trust. The second part, i.e., sufficiency, is shown by proposing a general solution to the fair exchange problem under the aforementioned condition.

2.1 Introduction

The notion of trust is central to the solvability of fair exchange, since, as seen in the previous chapter, it is impossible without at least one trusted process. Trust and its various possible settings in a distributed system are thus the focal points of this chapter, as illustrated in Figure 2.1. By introducing a trusted process available to all other processes, the *trusted third party* (TTP) setting produces an effective solution. Moreover, the TTP provides an unconditional solution with respect to the number of Byzantine processes. In other words, even if there is only a single correct process facing any number of Byzantine

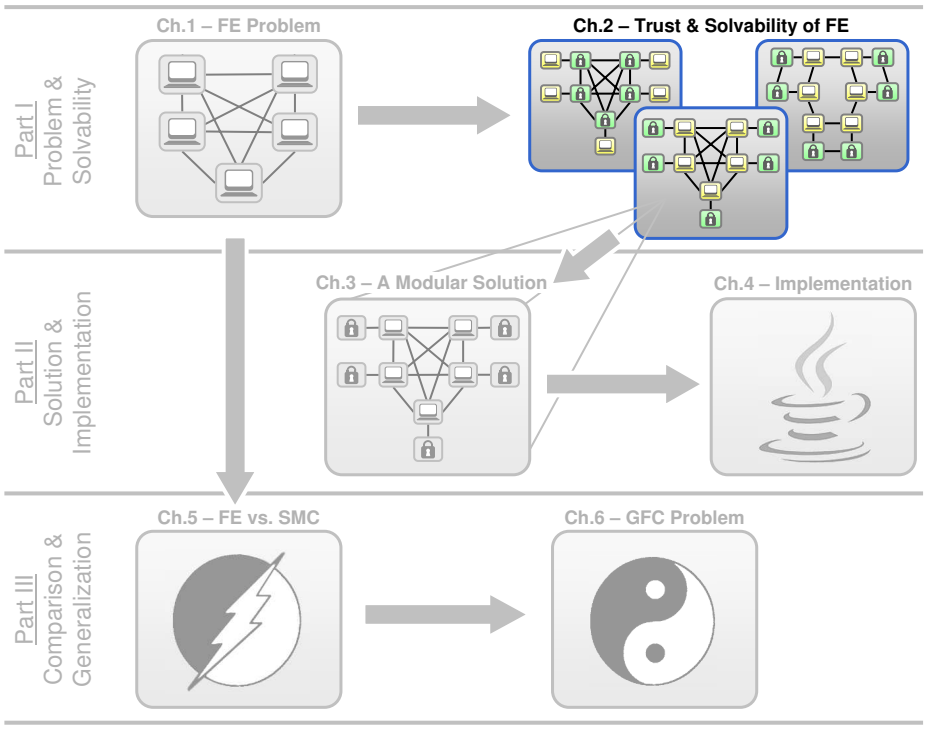


Figure 2.1: Thesis outline.

processes, *fairness* is ensured nonetheless. However, this solution relies entirely on a strong assumption, i.e., the reliability of the TTP.

Other approaches depart from the traditional TTP-based approach by relying on fully decentralized tamperproof modules [AGG+04, AGGV05, AV03], i.e., assuming fully connected processes but embedded tamperproof modules each dependent on its hosting process for communicating. These solutions work by having the tamperproof modules execute a specific algorithm solving a variant either of the consensus problem or of the atomic commitment problem. However, in this particular case, the solvability of fair exchange is conditioned by the number of Byzantine processes. True fair exchange is only possible if there is a majority of correct processes.

The role of trust in these two settings – the centralized TTP and the embedded tamperproof modules – is different in each but nonetheless critical in both in order to achieve *fairness*. The aim of this chapter is to provide a model of trust suiting as many settings as possible, including the two above, and to produce a general solvability condition in this context.

2.2 System Model: Adding Trust

Intuitively, our model consists of a synchronous distributed system composed of two types of process: *participants*, which are processes potentially subject to Byzantine failures, and *trustees*, which are known a priori to be correct (and which can thus be trusted). Adding only a single trusted process would however limit the scope of our model and imply a specific role for that trustee, i.e., that of a TTP. For this reason, we associate a trustee with each process, hence uniformly splitting the notion of trust among participants of the exchange. This enlarges the domain of topologies, to range from the TTP approach to fully decentralized settings.

More formally, as in Section 1.2, we consider a distributed system consisting of a set Π of n *processes*, $\Pi = \{p_1, \ldots, p_n\}$, and complete our model with a set Π' of n *trusted processes*, $\Pi' = \{p'_1, \ldots, p'_n\}$, i.e., known a priori to be correct by all other processes. To distinguish them, processes of Π are called *participants* and processes of Π' *trustees*. Moreover, each participant p_i is matched in a one-to-one relationship with at least its corresponding trustee p'_i. The set Π^+ is then the set of all $2n$ participants and trustees, i.e., $\Pi^+ = \Pi \cup \Pi'$. Participants are processes actually taking part in the exchange by offering and demanding items, and they may exhibit Byzantine behaviors. Trustees on the contrary are *trusted processes* that have no direct interest in the exchange. Their role is to decide when it is appropriate to provide their associated participant with its expected item.

We also assume the existence of a *Public Key Infrastructure* (PKI), which provides *sign* and *encrypt* primitives along with the corresponding *signature verification* and *decipher* primitives. However, the *signature verification* primitive is not used explicitly, since this is done inside functions that verify the validity of messages. Each process (participants and trustees) thus owns a private key and has made the corresponding public key accessible to all other processes. Among other things, this assumption provides message unforgeability.

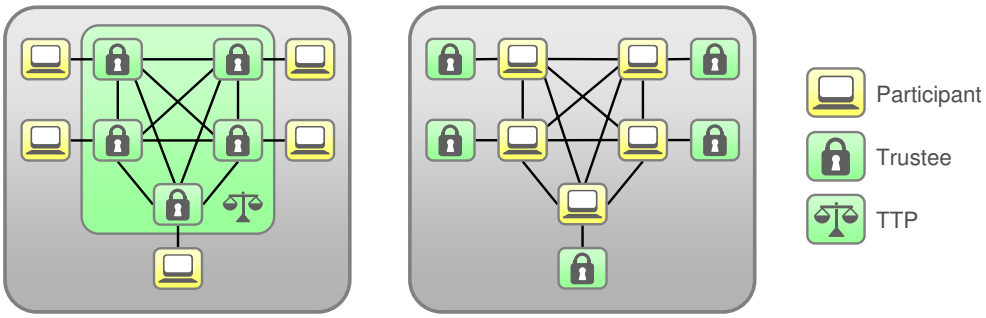
2.2.1 Topology and Synchrony.

Processes of Π^+ form a connected communication network and communicate by message passing. The system is *synchronous*: it exhibits *synchronous computation* and *synchronous communication*, i.e., there exist upper bounds on processing and communication delays, and these are known.[1] Links are reliable bidirectional communication channels, i.e., *perfect links* (PL). They provide *send* and *deliver* primitives (respectively PL.send() and PL.deliver()) and

[1] If delays exist but are unknown, they can be found through the use of adaptive timeouts.

ensure the *reliable delivery*, *no duplication* and *no creation* properties described in Section 1.2. The *synchronous* system assumption implies that the delivery will occur within some known time bound Δ_{PL}.

Regarding the network topology, we assume that processes of Π^+ form a connected graph and that there exists a direct link between any participant and its trustee. The notion of trustee allows us to produce a generic model applicable to various trust and network topologies. In particular, this model does not dictate the role of trustees in the fair exchange protocol, i.e., the amount of computation trustees bear or how they are connected to one another, except for the fact that each trustee is connected to its corresponding process.



(a) TTP-based topology. (b) Guardian angels topology.

Figure 2.2: Examples of valid topologies as defined in our model.

As a consequence, most existing solutions, either centralized or decentralized, can be described in our model. Simpler topologies, using fewer trustees, can easily be transformed to fit our model. Indeed, any trustee matched with several participants can be represented by a cluster of as many trustees, fully interconnected, with each trustee of this cluster matched with a single participant. For example in Figure 2.2(a), the classical centralized TTP setting, as in [ASW00], is transformed into a cluster of n fully interconnected trustees. Figure 2.2(b) illustrates a distributed trust setting, as with Guardian Angels [AGGV05].

2.2.2 Executions and Failure Patterns

The definitions of execution and failure given hereafter are the same as those introduced in Section 1.2. However, failures refer exclusively to participants, i.e., processes of Π, since trustees are correct by definition. In each step of an execution of an algorithm A, a process may thus (1) send a message, (2) receive a message and (3) update its local state. A *Byzantine process* is one that deviates from A in any way. A *Byzantine failure pattern* f is then defined as

a function from T to 2^{Π} where $f(t)$ denotes a set of Byzantine participants that have deviated from A through time t. A failure pattern f can thus be seen as a projection of all process failures during some execution of A. Once a process starts misbehaving, it cannot subsequently be considered correct, i.e., $f(t) \subseteq f(t+1)$.

Let $\mathsf{Byz}(f) = \bigcup_{t \in T} f(t)$ and $\mathsf{Cor}(f) = \Pi - \mathsf{Byz}(f)$ denote respectively the set of all Byzantine processes and the set of all correct processes in an execution with failure pattern f. We then define the set F_b of all failure patterns where no more than b processes are Byzantine. More formally, since n is the number of processes in Π, F_b is the largest subset of F such that, for any failure pattern f in F_b, $|\mathsf{Byz}(f)| \leq b$, with $0 \leq b \leq n$. That is, we have:

$$F_b = \{f \in F \ : \ |\mathsf{Byz}(f)| \leq b\} \text{ with } 0 \leq b \leq n \ .$$

From this definition, b is the maximum number of Byzantine processes in any failure pattern of F_b and $F_n = F$. Finally, we define the set $F_f^{\sim}$ of all failure patterns producing the same set of Byzantine processes as f. More formally, given some failure pattern f, $F_f^{\sim}$ is the largest subset of F such that, for any failure pattern f' in $F_f^{\sim}$, $\mathsf{Byz}(f') = \mathsf{Byz}(f)$. That is, we have:

$$F_f^{\sim} = \{f' \in F \ : \ \mathsf{Byz}(f') = \mathsf{Byz}(f)\} \ .$$

2.3 Solvability in the Model with Trust

Two opposite settings both provide solutions to the problem of fair exchange and can be described using our new model, as illustrated in Figure 2.3. In

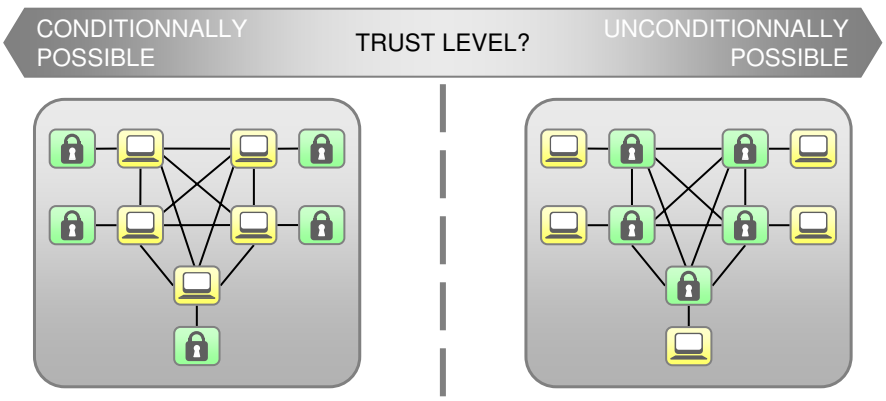
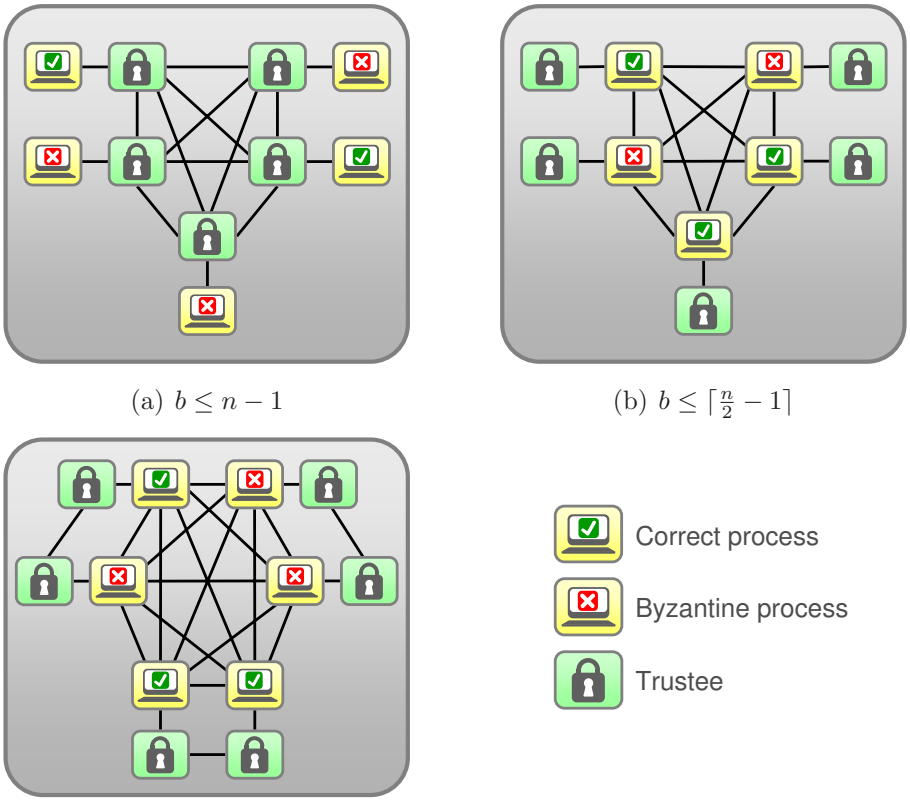


Figure 2.3: How much trust is needed for solving fair exchange?

one, the centralized TTP approach allows for solution to the fair exchange problem regardless of the number of Byzantine processes. In the other, the guardian angels approach also solves fair exchange but requires a majority of correct participants. Interestingly, this honest majority assumption is related to trust. So we have to address the question of the connection between these two approaches. In other words, can we express some general condition on the level of trust needed to solve the problem of fair exchange?

2.3.1 The Reachable Majority (RM) Condition

Intuitively, the *reachable majority* (RM) condition ensures that any correct participant process is reliably connected to a strict majority of trustees. To define the RM condition formally, we first define the notion of *reliable path*. Let p_i and p_j be two correct processes of Π^+. For any failure pattern f, we say that p_i and p_j are connected by a *reliable path* if there exists at least one path between p_i and p_j such that no process along that path is in $\mathsf{Byz}(f)$.



(a) $b \leq n - 1$

(b) $b \leq \lceil \frac{n}{2} - 1 \rceil$

(c) $b \leq \frac{n}{2}$

Figure 2.4: Topologies under the RM condition.

Furthermore, given a process $p_i \in \Pi$ (participants) and a failure pattern $f \in F$, we define $C^f_{p_i}$ as the largest subset of Π' (trustees) such that any trustee p'_j of $C^f_{p_i}$ is connected to p_i by a reliable path. The RM condition is then formally defined as follows.

Definition 2 (Reachable Majority Condition)**.** *Topological condition under which, for any correct process $p_i \in \Pi$, for any failure pattern $f \in F_b$ and even in the presence of up to b Byzantine processes, $|C^f_{p_i}| > \frac{n}{2}$.*

We then define the notions of *major* and *minor* trustees. Intuitively, in a given execution, a trustee is said to be *major* if it is connected to all correct processes by a reliable path. Otherwise, it is said to be *minor*.

Definition 3 (Major Trustee)**.** *Given a failure pattern f, a trustee p'_j is a major trustee if $p'_j \in C^f_{p_i}$, for any correct process $p_i \in \Pi$.*

The strict majority of Definition 2 ensures not only that in any given execution there are indeed some major trustees but also that there is a majority of them. Also note that if the RM condition is met, it implies that all correct processes and the majority of trustees are interconnected by reliable paths. However, it is important to understand that it neither implies nor requires a majority of correct processes.

Given a specific topology, the RM condition puts a constraint on the maximum number b of Byzantine processes. Figure 2.4 gives examples of topologies under the RM condition, with their resultant upper bounds on the number of Byzantine processes. As illustrated in Figure 2.4(a), a TTP allows any number of Byzantine processes, whereas Figure 2.4(b) and 2.4(c) show topologies only allowing, respectively, a minority and up to a half of processes to be Byzantine.

2.3.2 Impossibility Result

In Section 1.4, it was shown that fair exchange is impossible without at least one trusted process. On the other hand, a cluster of trustees acting as a TTP yields a solution. However, depending on the network topology, the presence of trustees is not sufficient. In the context of the model with trustees the *reachable majority* condition is necessary in order to solve fair exchange. This is the subject of Theorem 2 below, which relies on Lemma 1.

Lemma 1. *In any topology, if there exists an algorithm A solving fair exchange with up to b Byzantine processes, then for any failure pattern $f \in F_b$, there exists an execution associated with a failure pattern $f' \in F_f^{\sim}$ such that every process in $\mathsf{Cor}(f')$ releases its expected item.*

Proof. Consider an execution E of A in which all processes are correct. From the *non-triviality*, *termination* and *validity* properties, in E every process eventually releases its correct item. Now consider any failure pattern $f \in F_b$. From E, we derive an execution E' associated with a failure pattern $f' \in F_f^{\sim}$, i.e., $\mathsf{Byz}(f') = \mathsf{Byz}(f)$, such that, in E', every process of $\mathsf{Byz}(f')$ deviates from A just before releasing its item, e.g., by crashing. Since E' is indistinguishable from E for all correct processes, every process in $\mathsf{Cor}(f')$ releases its correct item. □

Theorem 2. *In the context of a synchronous model with trustees and Byzantine failures, there is no deterministic solution to the fair exchange problem, if the reachable majority condition is not satisfied, even in the presence of a single Byzantine process, i.e.,* $b = 1$.

Proof. The proof is by contradiction.
Assume that some algorithm A solves fair exchange and that the *reachable majority* condition is not satisfied, i.e., there is some correct process $p_i \in \Pi$ and some failure pattern $f \in F_b$ for which $|C_{p_i}^f| \leq \frac{n}{2}$, even for $b = 1$. From Lemma 1, we know that there exists an execution E' associated with a failure pattern $f' \in F_f^{\sim}$, such that every process in $\mathsf{Cor}(f')$ releases its expected item. Hence, in E', p_i receives its expected item, e.g., m_j, from its trustee p_i'. We now have to consider two cases: (a) the transmission of m_j from p_i' to p_i depends on the reception by p_i' of some message x sent by some trustee $p_j' \in \Pi' - C_{p_i}^f$, and (b) the transmission of m_j from p_i' to p_i is independent of the reception by p_i' of any message sent by any trustee $p_j' \in \Pi' - C_{p_i}^f$.[2]

Case (a). From E', we can derive an execution E'', where message x is blocked by some Byzantine process along the *unreliable* path between p_i' and p_j', as well as any following messages. Since E'' is indistinguishable from E' for any process unreliably connected to p_i', i.e., processes associated with trustees of $\Pi' - C_{p_i}^f$, these processes release their expected item in E''. However, in E'', p_i' never receives x. Since the transmission of m_j depends on the reception of x, p_i' never sends m_j to p_i. To satisfy the *validity* and *termination* properties, p_i releases φ but thus violates *fairness*. This leaves us with case (b).

Case (b). From E', we can derive an execution E''', in which some Byzantine process p_k fails to send the expected item to some trustee $p_j' \in \Pi' - C_{p_i}^f$, with p_j correct and $(p_k, p_j) \in \Omega$. Since the transmission of m_j from p_i' to p_i is independent of the reception of any message sent by any trustee of $\Pi' - C_{p_i}^f$ (including p_j'), for p_i' and p_i, executions E''' and E' are

[2] By definition, $C_{p_i}^f = C_{p_i}^{f'}$, for any failure f and f' such that $f' \in F_f^{\sim}$.

indistinguishable. Thus, in E''', p_i releases its expected item. However, since p'_j never receives the expected item of p_k, neither does p_j. To satisfy the *validity* and *termination* properties, p_j eventually releases φ but thus violates *fairness*. Therefore, algorithm A does not solve fair exchange. A contradiction. □

2.3.3 Solvability Result

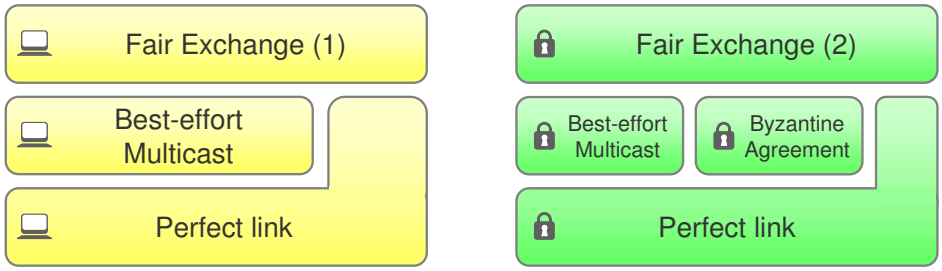
We have shown that the RM condition is *necessary* in order to solve the problem of fair exchange deterministically in the model with trustees. Theorem 3 below shows that it is also sufficient. This result allows us to apprehend precisely the solvability of the fair exchange problem for various topologies, possibly any topology. Indeed, given a topology and a number of Byzantine processes, one can infer whether a solution exists in that context. Perhaps more interestingly, it is possible to determine the maximum number of Byzantine processes that a specific network topology may sustain while still allowing true fair exchange, as opposed to probabilistic fair exchange, in which *fairness* is only ensured with a certain probability.

Theorem 3 (Solvability)**.** *In the context of a synchronous model with trustees and Byzantine failures, there exists a deterministic solution to the fair exchange problem under the reachable majority condition.*

Proof. In Section 2.4, we present Algorithms 2.1 and 2.2 (on pages 35 and 36), which combine to produce a generic solution to fair exchange, for any topology and any number of Byzantine processes satisfying the RM condition. Theorem 3 is then proven by showing that this solution preserves the *validity*, *uniqueness*, *non-triviality*, *termination*, *integrity* and *fairness* properties of fair exchange. □

2.4 Fair Exchange under the RM Condition

As described in Section 2.2 (system model), participants communicate by message passing and the network is a connected graph with respective participants and trustees connected directly. Algorithm 2.1, executed by correct participants, and Algorithm 2.2, executed by trustees, combine to compose our modular solution. As shown in Figure 2.5, the algorithms rely on three communication modules: a *best-effort multicast* module and a *Byzantine agreement* module described hereafter, and *perfect links*, as introduced in Section 1.2. Note that we merely aim at proving that a general solution does exist under the RM condition and are therefore not concerned with performance.



(a) Modules of a participant.

(b) Modules of a trustee.

Figure 2.5: Layered diagrams of the modules involved in our solution.

2.4.1 Best-effort Multicast

In order to solve fair exchange, Algorithms 2.1 and 2.2 rely on a *best-effort multicast* (BM) module that provides processes (participants and trustees) with the means to send messages to any group of processes with best effort. As described in our extended model, processes that are directly connected can communicate via reliable channels. However, two processes that do not benefit from a direct link are not necessarily connected reliably, since paths between them might go through a Byzantine process, making communications unreliable. The BM module provides a means of sending messages reliably to processes reachable through at least one reliable path.[3] The module provides two primitives, *send* and *deliver*, described hereafter.

BM.send$(p_i, S, \text{'TYPE'}, m)$ – Enables a process p_i to multicast a message m to a defined set S of processes. The message type prevents confusion between different messages.

BM.deliver$(p_i, p_j, \text{'TYPE'}, m)$ – Works as a callback and enables a process p_j of S to receive a message m from process p_i.

The best-effort multicast ensures the following set of properties.

No duplication. No message is delivered by a process more than once.

No creation. If a message m is delivered by some process p_j, then m was previously sent by some process p_i.

Termination. Let p_i and p_j be two correct processes connected by a reliable path. If p_i BM.sends a message m to a set S, with $p_j \in S$, then p_j eventually delivers m.

[3]This can be achieved using flooding as presented in Section 2.4.6 or some more sophisticated algorithm [DFS05].

Agreement. Let p_i and p_j be two correct processes of any set S that are connected by a reliable path. If p_i BM.delivers a message m BM.sent to S, then p_j BM.delivers m.

The *termination* property of BM is achieved within some maximum time bound Δ_{BM}, e.g., by having in the worst case $\Delta_{\text{BM}} = n \times \Delta_{\text{PL}}$, with Δ_{PL} being the delay of perfect links.

2.4.2 Byzantine Agreement

Algorithm 2.2 also relies on a *Byzantine agreement* (BA) module, which provides trustees with the means to reach agreement among major trustees, in spite of Byzantine failures that may occur along the various paths. The module is a particular application of the Byzantine agreement solution presented in [LSP82]. Section 2.4.7 discusses how this solution can be applied to our model with participants and trustees organized according to the topology of Figure 2.2(b). The BA module provides three primitives, BA.start(), BA.send() and BA.deliver().

BA.start(p'_j) – Enables a trustee p'_i to start an execution of BA in order to receive a message from a trustee p'_j. For each execution of the protocol, every trustee calls the *start* primitive at the same time (see discussion about the timing assumptions below) and trustee p'_j calls the *send* primitive.

BA.send(p'_i, m) – Enables a trustee p'_i to broadcast a message m reliably to all trustees.

BA.deliver(p'_i, M) – Works as a callback and enables a trustee p'_j to receive a set M of messages as the result of a reliable broadcast by trustee p'_i. Possible outcomes of the broadcast are twofold: (1) M is a singleton, meaning that transmissions from the sender were not blocked, so the message can be used; (2) M is the empty set, meaning that the sender did not call the *send* primitive in a timely fashion or that the messages from the sender were blocked.

The goal is to prevent Byzantine processes from threatening agreement among correct processes and to ensure the following interactive consistency (IC) properties.

IC1 – Agreement. If a major trustee BA.delivers a set of messages M, then every major trustee BA.delivers M.

IC2 – Validity. If a major trustee BA.sends a message m, then every major trustee *eventually* BA.delivers the set $\{m\}$.

When relying on unforgeable signed messages, a solution is known to exist for any number of Byzantine processes [LSP82]. However, while we indeed assume unforgeable signed messages, in our case, the number of Byzantine processes is still restricted by the reachable majority condition. It is nonetheless interesting to note that the use of such a Byzantine agreement module does not further restrict that number.

Timing Assumption. An implicit assumption in [LSP82] is that all trustees start at roughly the same time to allow the absence of messages to be detected. Since we assume that trustees start Algorithm 2.2 roughly at the same time, the *start* primitive enables us to ensure explicitly that missing messages are detected by having all trustees call the *start* primitive at the same time. This ensures the termination of BA, even if a trustee does not send any vote or if messages are blocked by some Byzantine processes. Also, in a synchronous model, an essential requirement of Byzantine agreement is that the delivery of a set of messages is achievable within a maximum time bound. Furthermore, in [DS83] it is shown that deterministic Byzantine agreement protocols with authentication have a lower bound $b + 1$ on the number of rounds, with b the number of Byzantine processes. Thus, if we choose such an implementation of BA, this time bound can be computed as a function of Δ_{BM}, the latency of BM, and b, the number of Byzantine processes, and hence $\Delta_{\mathrm{BA}} = (b + 1) \times \Delta_{\mathrm{BM}}$.

In [FM88, KK06], solutions to the Byzantine agreement problem allow expected time bounds that are constant, i.e., not correlated with the number of Byzantine processes, with the assumption of an honest majority. However, while the honest majority assumption suits our needs, these solutions do not achieve *deterministic* Byzantine agreement and are thus not usable for ensuring true fairness.

2.4.3 A General Algorithm of Fair Exchange

Algorithms 2.1 and 2.2 provide a general solution to the fair exchange problem for any topology and any number of Byzantine processes meeting the RM condition. We assume that all correct processes – including all trustees – have local clocks that are synchronized within some fixed maximum error, as discussed in [PSL80], so they are able to start the algorithms roughly at the same time. We also assume that the *upon* actions (found in both algorithms) are executed atomically with respect to one another. Participants execute Algorithm 2.1, which initiates the fair exchange protocol, and trustees execute

Algorithm 2.2. In Algorithm 2.1, each participant sends an encrypted version of its offered item to the trustee of the corresponding participant, according to Ω (the terms of the exchange). It then waits to receive and release the content of the first message sent by its trustee. The termination of Algorithm 2.1 is ensured by the timeout contained in Algorithm 2.2. In Algorithm 2.2, each trustee waits to receive the item expected by its associated participant. Algorithm 2.2 is then structured in two phases: (1) voting, and (2) clue exchange.

Algorithm 2.1 Fair exchange – Protocol executed by participant p_i

```
1: Uses: Perfect Link (PL), Best-effort Multicast (BM)
2: Initialisation:
3:     released ← FALSE

4: function offer(item, p_j)
5:     BM.send(p_i, {p'_j}, 'ITEM', encrypt(p'_j, item))        {send encrypted item to p'_j}

6: upon PL.deliver(p'_i, p_i, item) do                          {PL callback}
7:     if ¬released then                                        {check if not released}
8:         released ← TRUE                                      {set released to true}
9:         release(item)                                        {release the item received}
```

Voting phase. In this phase, trustee p'_i sends its vote to every trustee to inform them that it holds the expected item, and waits to receive the vote of every trustee. In Algorithm 2.2, once trustee p'_i receives the encrypted item (line 7), it deciphers it using its private key, checks if it matches its description and starts the voting phase. The trustee signs and broadcasts its PROCEED vote (line 12) using BA, indicating that it holds the expected item. It also starts BA for each trustee to ensure termination of all executions of BA. Then, upon reception of a vote, the validProceedVote() function checks if the delivered set is a singleton containing the PROCEED vote of the sender (line 17). If the vote is valid, it is added to the set of votes. Once all votes are gathered, a trustee knows that every trustee voted PROCEED and that they thus hold the expected item. With that information, trustee p'_i enters the final phase by signing and then sending the n votes – called the i-th clue – to every trustee (line 21).

Clue exchange phase. In this phase, trustee p'_i sends its clue to all trustees to inform them that it received all n votes, and waits to receive the clues from a majority of trustees (line 27).[4] Upon reception of a clue, the validClue()

[4]In certain topologies, this condition could be smaller, i.e., less than the majority. However, since we are merely providing a general solution and not trying to achieve efficiency, this condition is used because it is strong enough for all cases.

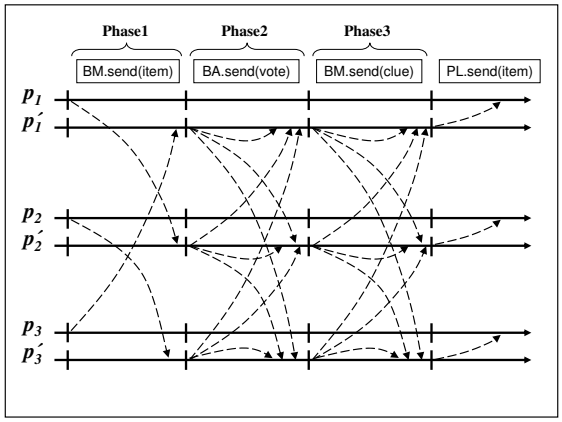
Algorithm 2.2 Fair exchange – Protocol executed by trustee p'_i

```
 1: Uses: Perfect Link (PL), Best-effort Multicast (BM), Byzantine Agreement (BA)
 2: Initialisation:
 3:     t_0 ← time()                                   {set t_0 to starting time}
 4:     d_i ← ...                                      {set description to known value}
 5:     item ← ⊥                                       {set variable to null}
 6:     votes, clues ← ∅                               {set variables to empty set}

 7: upon BM.deliver(p_j, p'_i, 'ITEM', sealedItem) do  {BM callback}
 8:     if (item = ⊥) then                             {check for duplicate send}
 9:         item ← decipher(sealedItem)                {decipher and store received item}
10:         if desc(item) = d_i then                   {check if item matches description}
11:             vote ← sign('PROCEED')                 {produce PROCEED vote}
12:             BA.send(p'_i, vote)                    {send vote}

13: upon time() > t_0 + Δ_BM do                        {item exchange phase is over}
14:     for all p'_j ∈ Π' do                           {for all trustees}
15:         BA.start(p'_j)                             {start BA}

16: upon BA.deliver(p'_j, vote) do                     {BA callback}
17:     if validProceedVote(vote) then                 {check vote}
18:         votes ← votes ∪ vote                       {add p'_j's vote to set}
19:         if (|votes| = n) then                      {if all votes are PROCEED}
20:             clue ← sign(votes)                     {produce clue}
21:             BM.send(p'_i, Π', 'CLUE', clue)        {send clue}
22:     else
23:         PL.send(p'_i, p_i, φ)                      {send φ to p_i}

24: upon BM.deliver(p'_j, p'_i, 'CLUE', clue) do       {BM callback}
25:     if validClue(clue) then                        {check if message is valid}
26:         clues ← clues ∪ {clue}                     {add p_j's clue to set}
27:         if (|clues| > n/2) then                    {check for majority of clues}
28:             PL.send(p'_i, p_i, item)               {send item to p_i}
```

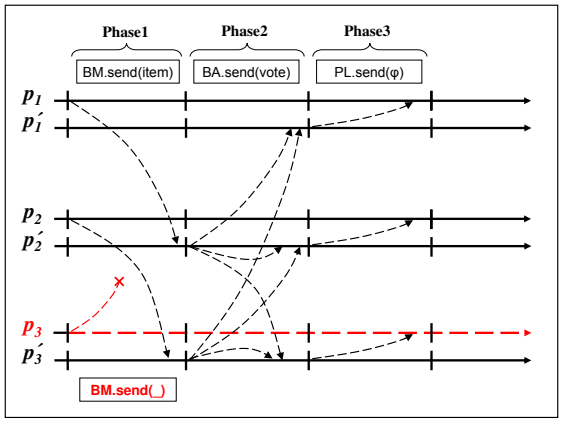
function checks if the clue contains a signed set of all n PROCEED votes (line 25). With $\lfloor \frac{n}{2}+1 \rfloor$ clues, it sends the deciphered item to its corresponding participant (line 28). The majority is necessary to ensure that at least one major trustee was able to produce its i-th clue in order for any process to release its item. At this stage, no Byzantine process is able to prevent trustees of correct processes from sending the expected item to their respective participant processes.

2.4.4 Examples of Executions

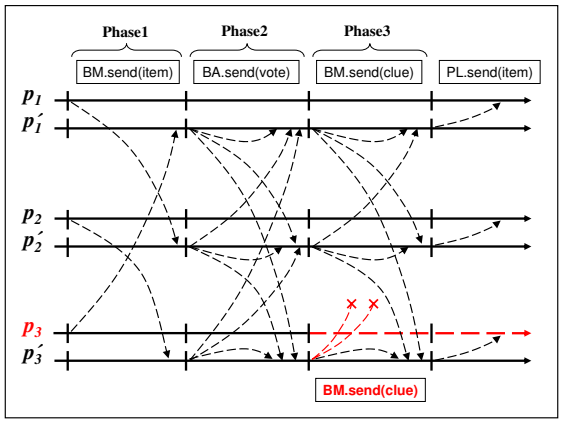
Figure 2.6 presents three possible executions of Algorithms 2.1 and 2.2 with three processes and up to one Byzantine process. In all figures, the top line shows the time line, the top labels corresponds to the correct behavior, the



(a) All three processes are correct.



(b) Process p_3 does not send its item.



(c) Process p_3 blocks the messages from its trustee.

Figure 2.6: Examples of executions of Algorithms 2.1 and 2.2.

arrows show the message transmissions and the bottom labels (if any) give the Byzantine behavior of process p_3.

Figure 2.6(a) shows an execution in which all processes are correct and they thus receive the expected item after the clue exchange phase. In Figure 2.6(b), process p_3 is Byzantine and deviates from the correct behavior by failing to send the item expected by p'_1. Thus p'_1 does not send its vote and eventually every correct process releases the abort item after the voting phase. However process p_3 is not able to release its item. In Figure 2.6(c), process p_3 is Byzantine and deviates from the correct behavior by blocking messages sent by its trustee p'_3 in the third phase, i.e., p'_3 cannot transmit its clues. However, in this case, all the correct processes are still able to release their items since they still receive a majority of clues.

2.4.5 Correctness Proof

In the following, we prove that Algorithms 2.1 and 2.2 solve fair exchange under the *reachable majority* condition by ensuring the *validity*, *uniqueness*, *non-triviality*, *termination*, *integrity* and *fairness* properties of fair exchange. Based on Lemma 2, the respective theorems hereafter validate each property. In the following, the term *process* is only used to designate participants, i.e., processes of Π, unless specifically mentioned otherwise.

Lemma 2. *If some trustee does not receive the expected encrypted item, then no trustee sends an item at line 28 of Algorithm 2.2.*

Proof. If some trustee does not receive the expected item, it does not send the PROCEED vote (line 12). Hence no trustee receives all n PROCEED votes, so no trustee sends its i-th clue. From the *no creation* property of perfect links, if no trustee sends its i-th clue, then no trustee receives any clue. Without a majority of clues, no trustee sends the item to its corresponding participant at line 28 of Algorithm 2.2. □

Theorem 4 (Validity)**.** *If a correct process p_i releases an item x, then either $x \in M$ and x matches d_i, or x is the abort item φ.*

Proof. In Algorithm 2.1, a process p_i only releases an item at line 9. Process p_i releases upon reception of an item from its trustee p'_i, so the possible items are those sent by p'_i in Algorithm 2.2. In Algorithm 2.2, trustee p'_i explicitly sends the abort item φ at line 23 so p_i would release φ. The only other case of item transmission is at line 28: p'_i sends the item that is stored in variable item. From Lemma 2, if a trustee sends an item at line 28, it has previously received the expected item and stored it in variable item. Since, from line 8,

no two different items can be stored in variable item, p'_i sends the expected item at line 28. Thus p_i would release the expected item. □

Theorem 5 (Uniqueness)**.** *No correct process releases more than once.*

Proof. The boolean variable released in Algorithm 2.1 and the atomic execution of *upon* statements prevent any correct process from releasing more than once. □

Theorem 6 (Non-triviality)**.** *If all processes are correct, no process releases the abort item φ.*

Proof. Since all processes are correct, each process sends the correct encrypted item at line 5 of Algorithm 2.1, as agreed in the terms of the exchange. From the *validity* property of BM, every trustee p'_i receives an item matching description d_i before time $t_1 = t_0 + \Delta_{\text{BM}}$, so every trustee produces and sends its PROCEED vote at line 12 of Algorithm 2.2 in a timely fashion. From the IC2 property of BA, no process receives an invalid PROCEED vote. Therefore, no trustee sends the abort item φ (line 23) of Algorithm 2.2 and thus no process releases φ. □

Theorem 7 (Termination)**.** *Every correct process eventually releases an item.*

Proof. The assumption that participants and trustees start Algorithms 2.1 and 2.2 at the same time and the timeout at line 13 of Algorithm 2.2 ensures that every trustee starts all n executions of BA at the same time. This implies that, from the existence of a time bound for the termination of BA and the IC1 property, there is a time after which either (a) every trustee of correct processes receives at least one invalid PROCEED vote and sends the abort item φ, prompting the corresponding correct process to release φ, or (b) every major trustee receives all n valid PROCEED votes. In the latter case, every major trustee produces and sends its i-th clue at line 21 of Algorithm 2.2. From the *validity* property of BM and the *reachable majority* condition, every trustee of correct processes receives a majority of clues and then sends the item at line 28 of Algorithm 2.2. Thus, from the *reliable delivery* property of perfect links, every correct process releases the item. □

Theorem 8 (Integrity)**.** *No process p_j releases an item m_i, with process p_i correct, if m_i matches description d_k of some correct process p_k, with $p_k \neq p_j$.*

Proof. Since any process p_k and its trustee p'_k are directly connected, no process p_j intercepts the transmission of any deciphered item m_i by p'_k at line 28 of Algorithm 2.2. Only in a single step of Algorithm 2.1, i.e., at line 5, does a correct process p_i transmit its item m_i through the network. Since p_i is

correct, p_i encrypts m_i using the public key of p'_k in order to send it through the network. Thus no process other than p_i and p_k holds a deciphered version of m_i and, since both are correct, they do not send a deciphered version of m_i to p_j. From the PKI unforgeability assumption, p_j is not capable of obtaining a deciphered version of m_i and therefore does not release m_i. □

Theorem 9 (Fairness)**.** *If any process p_i releases an item m_j matching description d_i, with p_i or p_j correct, then every correct process p_k releases an item matching its description d_k.*

Proof. The proof is by contradiction.
Assume that some correct process p_k does not release an item matching description d_k and that some other process p_i releases an item m_j matching description d_i, with p_i or p_j correct. If p_i releases m_j (line 9 of Algorithm 2.1), either p_i is correct and only releases an item received from its trustee p'_i; or p_j is correct and encrypted m_j before sending it to p'_i (line 5 of Algorithm 2.1) and thus p_i is only capable of releasing m_j by receiving it from its trustee p'_i. In either case, if p_i releases m_j, m_j is received from trustee p'_i, which sends m_j at line 28 of Algorithm 2.2. For this to happen, p'_i must have received a majority of clues in some previous steps. From the *reachable majority* condition, at least one of these clues is produced by some major trustee p'_x. Trustee p'_x therefore received all n PROCEED votes. Thus, from the IC1 property of BA, every major trustee also receives all n PROCEED votes, including all trustees of correct processes. This implies that no trustee of correct processes sends the abort item φ (line 23 of Algorithm 2.2), including p'_k, so p_k does not release φ. From the validity and termination properties of FE, if p_k does not release φ, then p_k releases an item matching description d_k. A contradiction. □

2.4.6 Best-effort Multicast: Solution and Proof

Algorithm 2.3 provides a solution to the *best-effort multicast* abstraction presented in Section 2.4.1 and therefore shows that the BM module is implementable in the context of our model. We assume that every process knows its direct neighbors and we define V_{p_i} as the set of neighbors of process p_i. Intuitively, Algorithm 2.3 satisfies the properties of best-effort multicast by having correct processes flood the network with multicasted messages. Flooding is achieved by forwarding any received message the first time it is received. Upon reception of a message, if the process is included in the set S of recipients, it also delivers the message. Note that, while flooding the network allows Byzantine processes the possibility of delivering messages illegitimately, it does not jeopardize the *validity* property of BM. Moreover, in our specific use of

Algorithm 2.3 Best-effort multicast protocol executed by process p_i

```
1:  Uses:
2:      Perfect Link (PL)

3:  Initialisation:
4:      forwarded ← ∅                                   {set of forwarded messages}

5:  function send(p_i, S, m)
6:      for all p_j ∈ V_{p_i} do                        {for all neighbors}
7:          PL.send(p_i, p_j, ⟨p_i, S, sign_i(m)⟩)      {sign and send the message}
8:      if p_i ∈ S then                                 {check if message destined to self}
9:          deliver(m)                                  {deliver the message}

10: upon PL.deliver(p_j, p_i, ⟨p_k, S, sign_k(m)⟩) do
11:     if m ∉ forwarded then                           {check if not forwarded}
12:         forwarded ← forwarded ∪ {m}                 {add the message to forwarded set}
13:         for all p_x ∈ V_{p_i} − {p_j} do            {for all neighbors except p_j}
14:             PL.send(p_i, p_x, ⟨p_k, S, sign_k(m)⟩)  {forward the message}
15:         if p_i ∈ S then                             {check if message destined to self}
16:             deliver(m)                              {deliver the message}
```

BM, this possibility of stealing messages has no impact on the outcomes of our fair exchange protocol.

Note that, in line 10 of Algorithm 2.3, the notation $\mathsf{sign}_k(m)$ in the *upon* action implies that the code is only executed if the message m is correctly signed by its originator p_k. In other words, it implies that the signature performed at line 7 is verified upon delivery of messages at line 10.

Correctness Proof

Our correctness proof, based on Lemma 3, shows that Algorithm 2.3 preserves the *agreement* and *termination* properties of best-effort multicast. It also indirectly shows that BM is implementable in our model.

Lemma 3. *Let p_i and p_j be any two correct processes that are connected by a reliable path. If p_i receives a message m, then p_j receives m.*[5]

Proof. The proof is by induction.
Basis step. Assume that some correct process p_i receives a message m (line 5 or 10) and that p_i and p_j are directly connected. Either (a) p_i is the originator of m and sends m to processes of V_{p_i} or (b) p_i receives m from some process p_x

[5]Note that in Lemma 3 the term 'receive a message' does not imply that the message is delivered but only that it is obtained either from the send() function (line 5 of Algorithm 2.3) or from the PL.deliver() callback (line 10 of Algorithm 2.3).

and sends m to processes of $V_{p_i} - \{p_x\}$. In both cases, from the *reliable delivery* property of perfect links, all processes of V_{p_i} eventually receive m. From our initial assumption that $p_j \in V_{p_i}$, p_j receives m.
Inductive step. Assume that any two correct processes p_i and p_j are connected by a reliable path. From definition of reliable paths, there exists a process p_k such that p_k is on that reliable path and $p_j \in V_{p_k}$. Moreover, p_k is correct and connected to p_i through a reliable path. Now – *inductive hypothesis* – assume that p_i and p_k receive a message m. Thus, either (a) p_k is the originator of m and sends m to processes of V_{p_k} or (b) p_k receives m from some process p_y and sends m to processes of $V_{p_k} - \{p_y\}$. In both cases, from the *reliable delivery* property of perfect links, all processes of V_{p_k} eventually receive m. From our initial assumption that $p_j \in V_{p_k}$, p_j receives m. □

Theorem 10 (Termination)**.** *Let p_i and p_j be two correct processes connected by a reliable path. If p_i sends a message m to a set S, with $p_j \in S$, then p_j eventually delivers m.*

Proof. Assume that two correct processes p_i and p_j are connected by a reliable path and that p_i sends a message m to a set S, with $p_j \in S$. Thus p_i receives m, as the originator of m. From Lemma 3, p_j also receives m. Since p_j is a correct process of S, either m is in the forwarded set of p_j and p_j has delivered m, or m is not in the forwarded set of p_j and p_j delivers m at line 16. □

Theorem 11 (Agreement)**.** *Let p_i and p_j be two correct processes of any set S that are connected by a reliable path. If p_i delivers a message m sent to S, then p_j delivers m.*

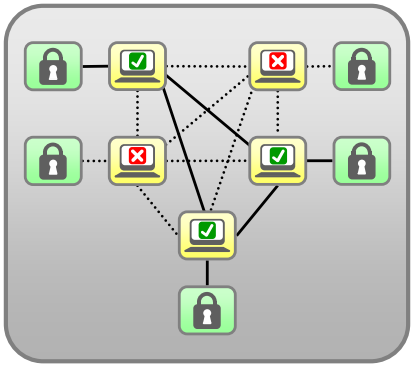
Proof. Assume that two correct processes p_i and p_j of S are connected by a reliable path and that p_i delivers a message m. Thus p_i receives m in a previous step of Algorithm 2.3 (line 5 or 10). From Lemma 3, p_j also receives m. Since p_j is a correct process of S, either m is in the forwarded set of p_j and p_j has delivered m, or m is not in the forwarded set of p_j and p_j delivers m at line 16. □

2.4.7 Implementation of BA in our Model

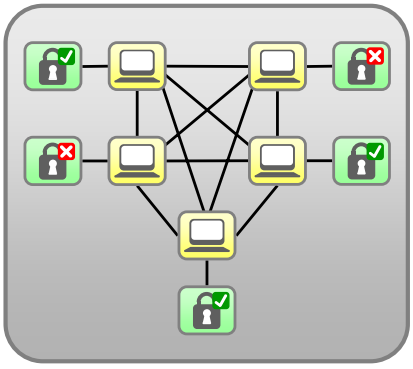
In [LSP82], the model is that of a fully connected distributed system, assuming reliable connections and Byzantine failures. The Byzantine agreement protocol is thus executed by a set of processes, some of which may exhibit Byzantine behaviors. In Section 2.4.2, we propose to apply this same protocol to our extended model, i.e., with processes and trustees. However, the network is not necessarily fully connected. In our case, the Byzantine agreement is executed

by the trustees, which by definition are all correct but not necessarily connected reliably. In spite of these differences, we claim that the problem and solution of [LSP82] can easily be adapted to our model.

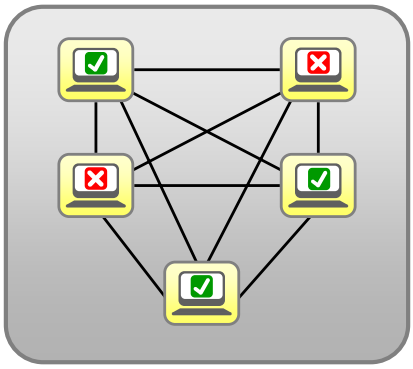
While communications along unreliable paths may be blocked by Byzantine processes, major trustees are connected reliably by definition, since they have at least one path connecting each other that does not go through a Byzantine process. Figure 2.7 illustrates, in a simple example, the transformation required in order to adapt our context to that of Byzantine agreement. Since assumptions in [LSP82] require reliable connections, we first adapt our context by considering that minor trustees are responsible for Byzantine failures happening along the unreliable paths leading to them. Thus our minor trustees correspond to the Byzantine processes of [LSP82] and our unreliable paths can now be considered reliable. Recall that, in our case, agreement only needs



(a) Step 1: A possible setting for Algorithm 2.2, with unreliable paths shown as dotted lines.



(b) Step 2: Conveying the responsibility of Byzantine behaviors onto minor trustees.



(c) Step 3: The resulting setting matching the original model of [LSP82]

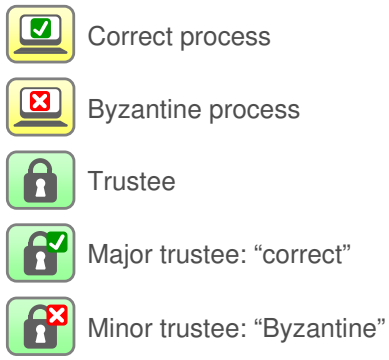


Figure 2.7: Three steps for adapting our context to the original setting of Byzantine agreement.

to be reached among major trustees, regardless of minor trustees, just as correct processes reach agreement in [LSP82] in spite of Byzantine processes. To summarize, Byzantine agreement can thus be applied to our context by considering major trustees as correct processes, minor trustees as Byzantine processes, and unreliable paths as reliable.

2.5 Revisiting Existing Solutions

In the light of our model with trustees, we propose to revisit two existing solutions presented in the introduction to this chapter to see how they are affected by the reachable majority condition.

2.5.1 The Trusted Third Party

Several algorithms described in the literature rely on the TTP paradigm. The simplest TTP-based algorithm consists in having processes send their items to a centralized trustee, the TTP. The TTP verifies that the terms of the exchange are respected and, if this is the case, forwards the items. These TTP-based solutions naturally fit our model with trustees. Our model uses n trustees instead of only one in TTP-based solutions. Mapping the TTP model to ours is done by considering all n trustees jointly as a fully connected cluster playing the role of the TTP. The network topology of this solution is such that each process is directly connected to one distinct trustee of the cluster, as illustrated in Figure 2.8. It is then fairly obvious to see that the TTP topology is so secure that the reachable majority condition is satisfied for any number of Byzantine processes.

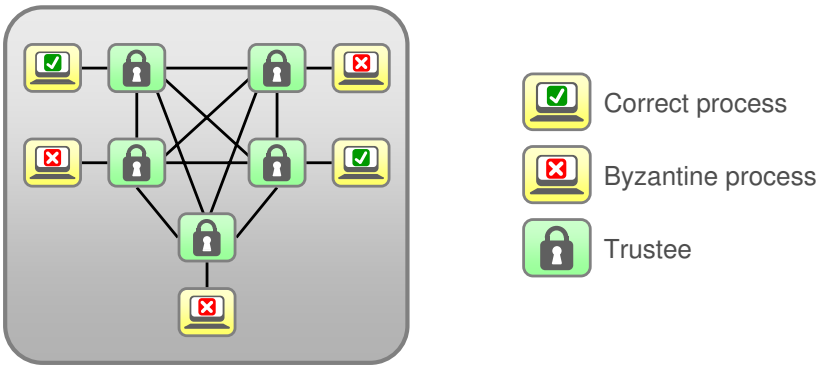


Figure 2.8: The TTP topology allows any number of Byzantine processes.

2.5.2 The Guardian Angels

In [AGGV05, AV03], guardian angels are defined as tamperproof security devices that are considered correct. Processes are fully interconnected by a communication network with bidirectional reliable channels. There are n guardian angels but each of them is only connected to one process. In other words, each process can directly communicate with its assigned security device but needs to go through some untrusted process to communicate with other security devices. Intuitively, in order to solve fair exchange, each item is encrypted and sent to the security device of the corresponding process, i.e., the process expecting the item. Security devices then enter a synchronization protocol, which upon success enables the devices to send the items to the processes. The assumption is made that the security devices are able to check the validity of the items and to encrypt messages. In a model with no upper bound on the number of Byzantine processes, the solution given solves fair exchange with a certain probability. The authors of [AGGV05, AV03] also show that, even in a synchronous model with security devices, no deterministic algorithm solves fair exchange without an honest majority, i.e., without $b < \frac{n}{2}$.

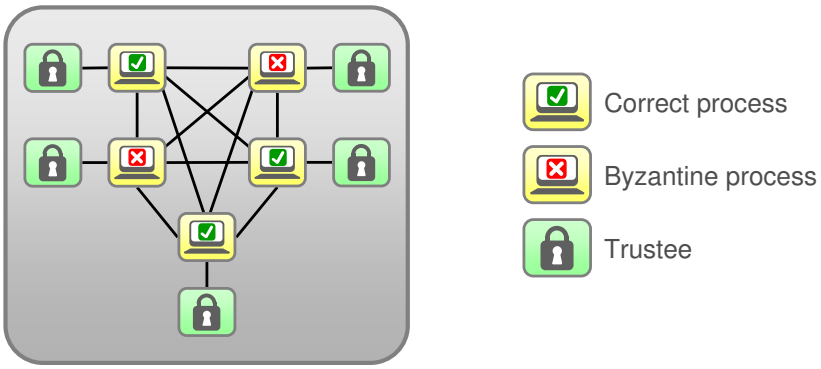


Figure 2.9: The Guardian Angels topology requires an honest majority.

The guardian angels approach fits our extended model if each of the n trustees represents one distinct security device. Theorem 2 tells us that if any process p is not connected through a trusted path to a majority of trustees, there is no solution to the problem. Accordingly, since each trustee is behind a distinct process, which is potentially Byzantine, there must be a majority of correct processes, as shown in Figure 2.9. From Theorems 2 and 3, we can then say that the guardian angels approach deterministically solves fair exchange if and only if there is a majority of correct processes. As one would expect, this result concurs with that found in [AGGV05]. Nonetheless, by introducing *dummy* messages within their fair exchange protocol executed by the guardian angels, an interesting feature of their solution lies in its ability to degrade its quality of service *gracefully* from deterministic to probabilistic fairness. While

the probability of violating fairness has an inversely proportional impact on the average complexity of the algorithm, it ensures that the probability of unfairness can be made arbitrarily low.

◊◊◊

Part II

Solution & Implementation

Chapter 3

A Modular Solution

If a problem has a solution,
there is no need to worry.
If a problem has no solution,
worrying will not help.

Tibetan Proverb

Abstract. This chapter proposes to focus on a specific network topology in order to provide a fully decentralized, yet realistic, solution to fair exchange. The aim is thus to optimize the general solution presented in the previous chapter by limiting the role of the trustees as much as possible. Our algorithm is based on three building blocks: a perfect link module, a secure box module and a module solving the well-known Byzantine agreement problem. The secure box modules, which play the role of trustees, are tamperproof but need not communicate directly with each other and are only required in a limited number of key steps of our algorithm. These features have the advantage of providing a more realistic decentralized solution in order to envisage applications in the real world.

3.1 Introduction

As showed in Chapters 1 and 2, solving fair exchange is impossible in the total absence of trust and remains problematic in most topologies supported in the model with trustees. The general solution given as a proof of sufficiency of the reachable majority condition is an important step in the study of the solvability of fair exchange. However, the generality of its topology assumption and the

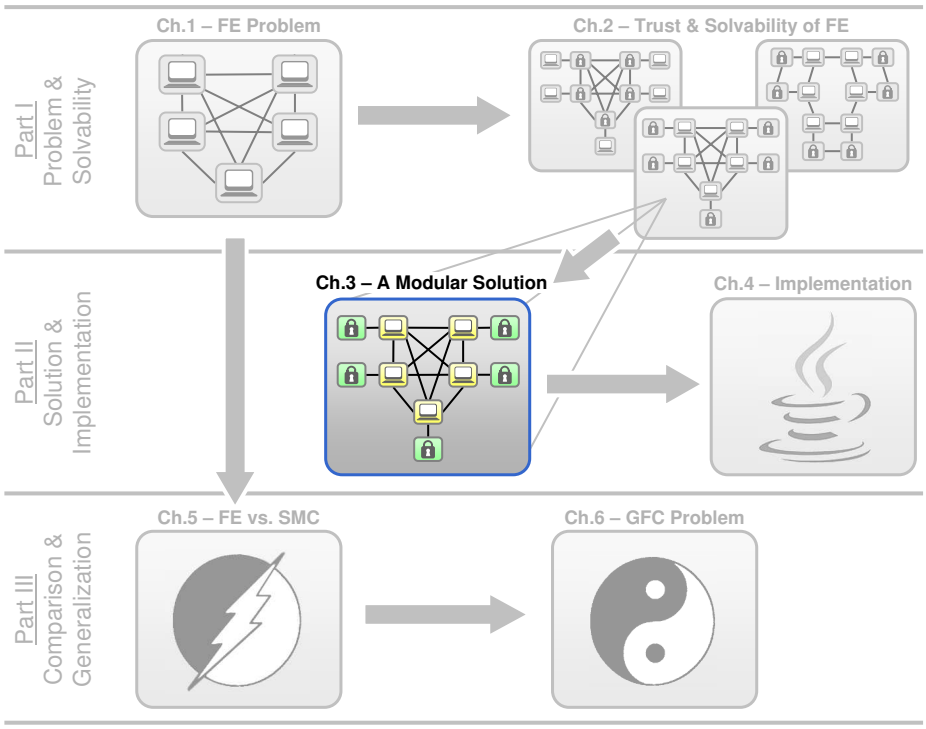


Figure 3.1: Thesis outline.

fact that it relies heavily on the computational power of the trustees does not allow for envisaging a concrete deployment of a fully decentralized solution. We propose to address this issue by providing a solution set in the context of a specific topology of the model with trustees, as illustrated in Figure 3.1. Our solution relies on a set of embedded trusted modules, one per peer participating in the exchange, each playing the role of a trustee. Note that this setting does not remove all the complexity of the problem since such embedded trusted modules must use unreliable channels in order to communicate with each other.

By relying on fully-decentralized tamperproof modules, our approach departs from the usual TTP-based approach and bears similarities to the guardian angels approach [AGG+04, AGGV05, AV03], which also assumes embedded tamperproof modules and a similar network topology. If this approach is similar to ours from the model perspective, it differs in the power given to the tamperproof modules. In the guardian angels approach, the tamperproof modules execute a specific algorithm solving either a variant of the consensus problem or of the atomic commitment problem. Our approach on the contrary tries to minimize the role of trusted modules to key steps of our algorithm, which is then executed outside these modules. This minimal approach makes

it easier to envisage implementations of our secure module in real hardware. Nonetheless, an interesting feature of the approach proposed in [AGGV05] is its ability to degrade its quality of service gracefully from deterministic to probabilistic fairness.

From a practical perspective, a trusted module is typically implemented as a tamperproof piece of hardware embedded in each peer host, e.g., a specialized chip or a smart card. In the industry, hardware-based solutions are gaining momentum, as illustrated by efforts from IBM, with both its PCI 4758 and PCI-X 4764 cryptographic coprocessors [DLP+01], and from Intel, with its Trusted Platform Module [Baj02]. Such solutions are expected to become mainstream, as the urge to go beyond the limits of software-based security increases, in particular in the realm of digital rights management.

Despite the fact that some of these specialized chips can be quite powerful, our approach tries to minimize the requirements on the conceptual trusted modules we consider, in order to facilitate the fulfillment of these requirements by current and future chips. Indeed, a fair exchange protocol should make minimal assumptions on the underlying hardware in order to support a wide range of platforms. Another important aim of our approach consists in providing a modular solution to fair exchange, where each element of the solution can easily be replaced by an alternative implementation.

3.2 System Model: a Specific Topology

Except for the topological aspects, which are discussed in Section 3.2.1, the system model is identical to the extended model of the previous chapter. Accordingly, we consider a distributed system consisting of a set Π of n participants, $\Pi = \{p_1, \ldots, p_n\}$ and a set Π' of n trustees, $\Pi' = \{p'_1, \ldots, p'_n\}$. Each trustee p'_i is matched in a one-to-one relationship with the corresponding participant p_i. The set Π^+ is then the set of all $2n$ processes, i.e., $\Pi^+ = \Pi \cup \Pi'$. Participants are processes actually taking part in the exchange by offering and demanding items, and they may exhibit Byzantine behaviors. Trustees on the contrary are *trusted processes* that have no direct interest in the exchange.

We also assume the existence of a *Public Key Infrastructure* (PKI), which provides *sign* and *encrypt* primitives along with the corresponding *signature verification* and *decipher* primitives. However, the last two are hidden inside functions that respectively verify the validity of messages and enable encrypted items to be verified and unsealed. Each process (participants and trustees) thus owns a private key and has made the corresponding public key accessible to all other processes. Among other things, this assumption provides message unforgeability.

3.2.1 Topology and Synchrony.

Processes of Π are fully interconnected by a communication network and communicate by message passing. Trustees are embedded in their respective processes and communicate by invoking primitives. The system is *synchronous*: it exhibits *synchronous computation* and *synchronous communication*, i.e., there exist upper bounds on processing and communication delays. We also assume the existence of some global real time clock, whose tick range, noted T, is the set of natural numbers.

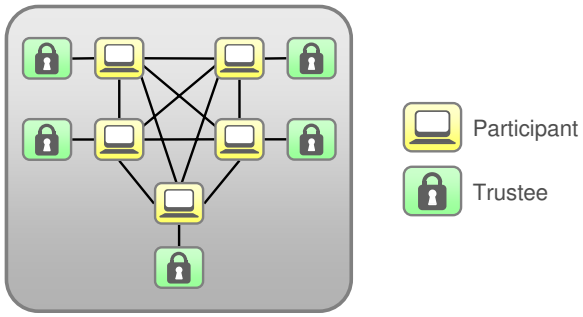


Figure 3.2: Topology with five participants and their trustees.

Regarding the network topology, we assume specifically that all processes of Π are fully interconnected but that trustees are only connected to their respective processes. Figure 3.2 illustrates such a topology with five participant processes and five trustees. Links are reliable bidirectional communication channels, i.e., *perfect links* (PL). They provide *send* and *deliver* primitives (respectively PL.send() and PL.deliver()) and ensure the *reliable delivery*, *no duplication* and *no creation* properties described in Section 1.2. The *synchronous* system assumption implies that the delivery will occur within some known time bound Δ_{PL}.

3.2.2 Executions and Failure Patterns

The definitions of execution and failure are as given in Section 1.2. Here, however, failures refer exclusively to participants, i.e., processes of Π, since trustees are correct. In each step of the execution of an algorithm A, a process may thus (1) send a message, (2) receive a message and (3) update its local state. A *Byzantine process* is one that deviates from A in any way. A *Byzantine failure pattern* f is then defined as a function from T to 2^{Π} where $f(t)$ denotes a set of Byzantine participants that have deviated from A through time t. A failure pattern f can thus be seen as a projection of all process failures

during some execution of A. Once a process starts misbehaving, it cannot subsequently be considered correct, i.e., $f(t) \subseteq f(t+1)$.

3.3 Fair Exchange with Secure Boxes

Our fair exchange protocol relies on three building blocks: a *perfect link* module, a *Byzantine agreement* module (BA) and a *secure box* module (SB), the latter playing the role of the trustee.[1] As suggested in Figure 3.3, only the code of the secure box module is tamperproof, whereas other modules may exhibit Byzantine behaviors. Compared to the solution of Section 2.4, we see that, by the limitation of their communication abilities, trustees are reduced to being simple embedded tamperproof modules. Furthermore, in this solution, we aim at minimizing the intelligence required from the secure boxes by moving as much of the computation as possible into the non-tamperproof part of the process.

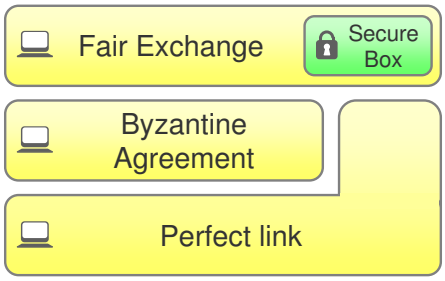


Figure 3.3: A layered diagram of the modules involved in our solution.

3.3.1 Secure Box

A secure box (SB) module is a simple tamperproof device and corresponds to the notion of trustee of our specific model. In this model, each trustee is only connected to its corresponding process, so the SB module must rely completely on its host in order to communicate with other trustees or processes. For example, a Byzantine host may isolate its trustee by blocking any incoming or outgoing information. In our solution, the role of the secure box is reduced as far as possible, i.e., it is completely passive and does not interact with the communication layer. A participant communicates with it by directly invoking primitives. The secure box can be seen as a local service available to

[1] In the following, both the terms *secure box* and *trustee* are used indifferently.

each participant. Combined with a public key infrastructure (PKI), its role is merely one of a safe. The SB device offers the following set of primitives.[2]

SB.isValidItem(m_j,d_i) – Returns a boolean value stating whether the encrypted item m_j matches description d_i.

SB.unseal(m_j,proof) – Returns the deciphered item m_j, if the proof of fairness, i.e., a majority of clues issued by other processes, is valid.

3.3.2 Byzantine Agreement

This module provides processes with a means of broadcasting messages reliably, in spite of Byzantine failures. While similar to the Byzantine agreement (BA) module introduced in Section 2.4 (page 31), it is used in the classical setting of [LSP82], i.e., the module is executed by processes of which some may be Byzantine. The main difference is that a Byzantine General[3] may lie by sending contradictory messages, so the set of messages delivered may contain more than one value. Since the *deliver* primitive is slightly different from the one introduced previously, we recall all three primitives provided by the BA module.

BA.start(p_j) – Enables a process p_i to start an execution of BA in order to receive a message from a process p_j. For each execution of the protocol, every correct process calls the *start* primitive at the same time, and process p_j then calls the *send* primitive.

BA.send(p_i,m) – Enables a process p_i to broadcast a message m reliably to all processes.

BA.deliver(p_j,M) – Works as a callback and enables a process p_j to receive a set S of messages as the result of a reliable broadcast. Possible outcomes of the broadcast are threefold: (1) M is a singleton, if the sender behaved correctly; (2) M contains more than one message, if the sender behaved incorrectly by sending different messages to different processes; (3) M is the empty set, if the sender did not send anything.

The goal is to prevent Byzantine processes from threatening agreement among correct processes and ensure the two interactive consistency (IC) properties. The properties of Byzantine agreement and the timing assumptions are unchanged from what was presented in the previous chapter (Section 2.4). Also,

[2]Both primitives use the *decipher* primitive of PKI.
[3]The General is the process broadcasting the message.

when relying on unforgeable signed messages, a solution is known to exist for any number of Byzantine processes [LSP82]. However, that number is still restricted by the reachable majority condition.

3.3.3 A Specific Algorithm of Fair Exchange

Algorithm 3.1 presents our solution to the fair exchange problem. We assume that all correct processes have local clocks that are synchronized within some fixed maximum drift, as discussed in [PSL80], so they are able to start the algorithm roughly at the same time. We also assume that *upon* actions (found in Algorithm 3.1) are executed atomically with respect to one another. Our algorithm is divided into three phases: (1) item exchange, (2) voting and (3) clue exchange.

Item exchange phase. The first phase of the algorithm allows every process to send the item it is offering. Each process sends its item to one and only one other process (line 11), as defined in the terms of the exchange. The item has to be encrypted, since the receiving process must not be able to have direct access to it. The encryption is thus made using the public key of the secure module of the receiving process. The secure box acts as a safe so that the receiving process only has access to the item at the end of the protocol.

Voting phase. In this phase, process p_i sends its vote to every process to inform them that it holds the expected item, and waits to receive the vote of every process. Once p_i receives the encrypted item, it asks its secure module to assert that the item matches the description (line 15) and starts the voting phase. The process signs and broadcasts its PROCEED vote (line 17) using BA, indicating that it has received the expected item. It also start BA for each process in order to synchronize with all the other correct processes (line 20). Then, upon reception of a vote, the validProceedVote() function checks if the delivered set is a singleton containing the valid PROCEED vote of the sender (line 22). If the vote is valid, it is added to the set of votes. Once all votes are gathered, a process knows that every process has voted PROCEED and has thus received the correct item. With that information, process p_i signs and sends the n votes – called the i-th clue – to every process (line 27). This is necessary in order to enter the final phase, which consists in having processes exchange their clues. Note that nothing prevents some Byzantine process p_k from producing its vote and its clue without having previously received its item. However such behavior cannot prejudice any process other than p_k.

Algorithm 3.1 Fair exchange protocol executed by process p_i

```
1: Uses:
2:     Perfect Link (PL), Byzantine Agreement (BA), Secure Box (SB)

3: Initialisation:
4:     t_0 ← time()                                   {set t_0 to starting time}
5:     d_i ← ...                                      {set description to known value}
6:     released ← FALSE                               {set variable to false}
7:     sealedItem ← ⊥                                 {set variable to null}
8:     votes, clues ← ∅                               {set variables to empty set}

9:  function offer(m_i, p_r)
10:    item ← encrypt(m_i, p'_r)                      {encrypt m_i using public key of p'_r}
11:    PL.send(p_i, p_r, 'ITEM', item)                {send encrypted item to p_r}

12: upon PL.deliver(p_s, p_i, 'ITEM', item) do        {PL callback}
13:    if (sealedItem = ⊥) then                       {check for duplicate send}
14:        sealedItem ← item                          {store received item}
15:        if SB.isValidItem(sealedItem, d_i) then    {ask SB to check item}
16:            vote ← sign('PROCEED')                 {produce PROCEED vote}
17:            BA.send(p_i, vote)                     {send vote}

18: upon time() > t_0 + Δ_PL do                       {item exchange phase is over}
19:    for all p_j ∈ Π do                             {for all processes}
20:        BA.start(p_j)                              {start BA}

21: upon BA.deliver(p_j, vote) ∧ (¬released) do       {BA callback and not released}
22:    if validProceedVote(vote) then                 {check vote}
23:        votes ← votes ∪ vote                       {add p_j's vote to set}
24:        if (|votes| = n) then                      {if all votes are PROCEED}
25:            for all p_k ∈ Π do
26:                clue ← sign(votes)                 {produce clue}
27:                PL.send(p_i, p_k, 'CLUE', clue)    {send clue}
28:    else
29:        released ← TRUE                            {set variable to true}
30:        release(φ)                                 {release φ}

31: upon PL.deliver(p_j, p_i, 'CLUE', clue) ∧ (¬released) do   {PL callback and not released}
32:    if validClue(clue) then                        {check if message is valid}
33:        clues ← clues ∪ {clue}                     {add p_j's clue to set}
34:        if (|clues| > n/2) then                    {check if majority of clues}
35:            released ← TRUE                        {set variable to true}
36:            item ← SB.unseal(sealedItem, clues)    {unseal item using SB}
37:            release(item)                          {release the unsealed item}
```

Clue exchange phase. In this phase, process p_i sends its clue to every process and waits to receive the clues from a majority of processes (line 34). Upon reception of a clue, the validClue() function checks if the clue contains a signed set of all n PROCEED votes (line 32). With at least $\lceil \frac{n}{2} + 1 \rceil$ clues, it can ask its secure module to release the sealed item by deciphering it using its private key (line 36). The majority is necessary to ensure that at least one correct process was able to produce its i-th clue in order for any process to release its item. At this stage, correct processes should be able to release without the help of any Byzantine process. Contrariwise, Byzantine processes should not be able to release without the help of at least one correct process. This aspect of our algorithm is further discussed in Section 3.4.

3.3.4 Examples of Executions

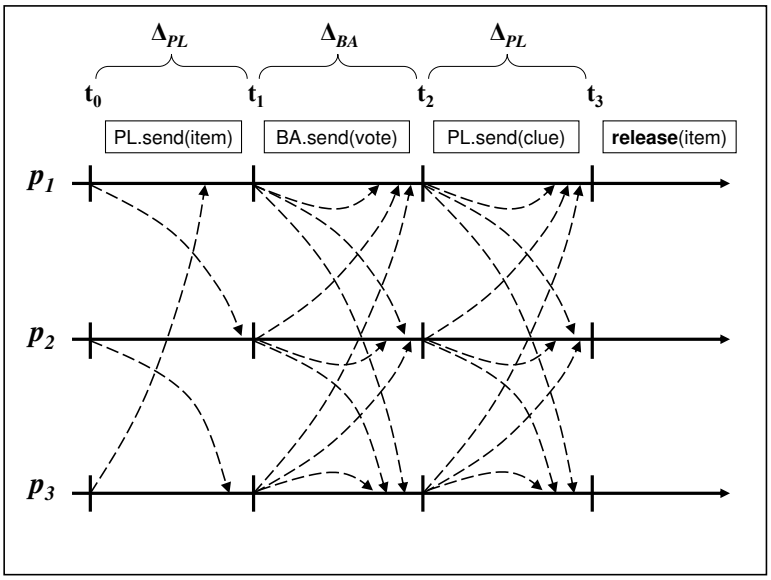
Figures 3.4 and 3.5 present four possible executions of Algorithm 3.1 with three processes and up to one Byzantine process. In all figures, the top line shows the time line with the different time bounds, the top labels correspond to the correct behavior, the arrows show the message transmissions and the bottom labels (if any) give the Byzantine behavior of process p_3.

Figure 3.4(a) shows an execution where all processes are correct. In Figure 3.4(b), process p_3 is Byzantine and deviates from the correct behavior by failing to send its clue. However, in this case, all the correct processes are still able to release their items.

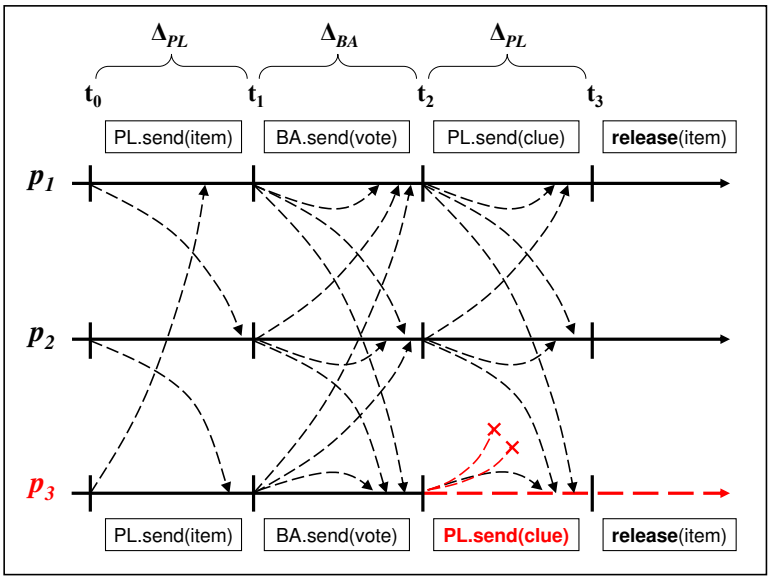
In Figure 3.5(a), process p_3 is Byzantine and deviates from the correct behavior by failing to send the item expected by p_1. Thus p_1 does not send its vote and eventually every correct process releases the abort item φ. Yet p_3 is not able to release its item. In Figure 3.5(b), process p_3 is Byzantine and deviates from the correct behavior by failing to send its vote or by sending it too late. Thus eventually every correct process release the abort item φ. Yet p_3 is not able to release its item.

3.3.5 Correctness Proof

In the following, we prove that Algorithm 3.1 solves fair exchange in the presence of b Byzantine processes, with $b < \frac{n}{2}$, by preserving the *validity*, *uniqueness*, *non-triviality*, *termination*, *integrity* and *fairness* properties of fair exchange. The restriction on the number of Byzantine processes is called the *honest majority* assumption. Based on Lemma 4, the respective theorems hereafter validate each property. In the following, the notation p'_i describing a trustee or a secure box is equivalent to that of SB used in Algorithm 3.1.

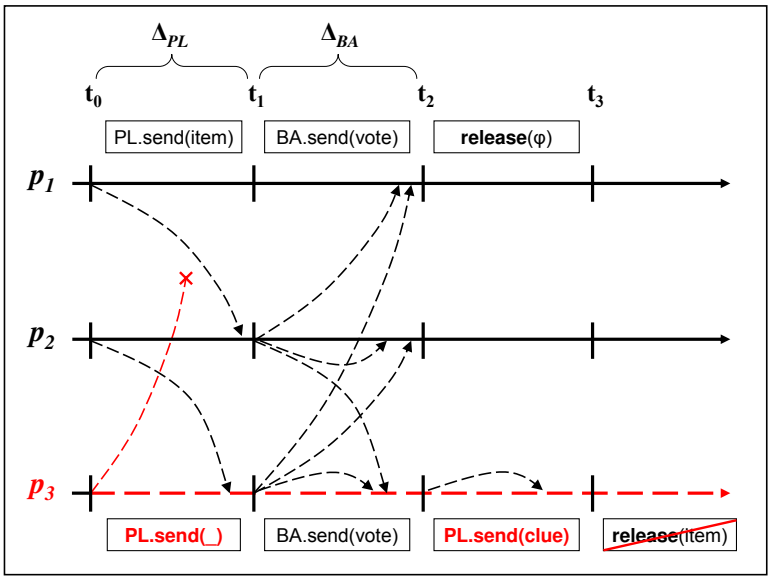


(a) All three processes are correct.

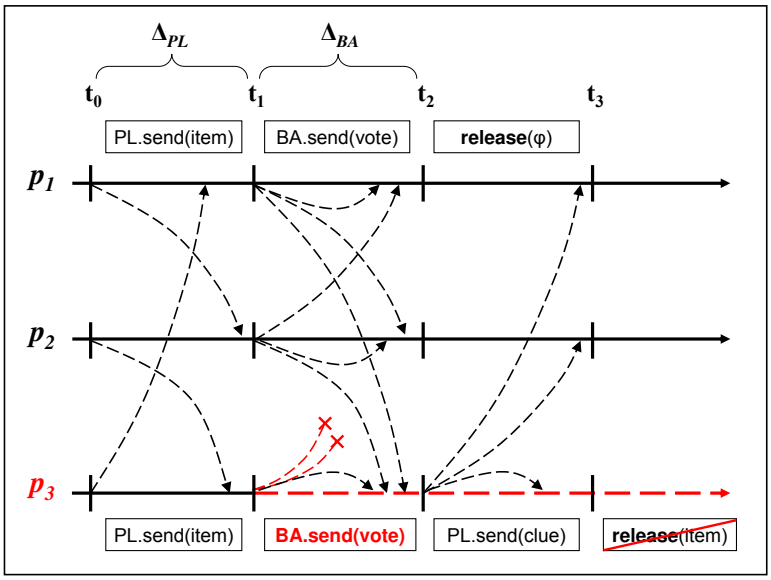


(b) Process p_3 deviates from Algorithm 3.1 by not sending its clue.

Figure 3.4: Successful executions of Algorithm 3.1.



(a) Process p_3 deviates from Algorithm 3.1 by not sending its item.



(b) Process p_3 deviates from Algorithm 3.1 by not sending its vote.

Figure 3.5: Aborted executions of Algorithm 3.1.

Lemma 4. *If some correct process p_j does not receive an encrypted item matching description d_j, then no process releases at line 37 of Algorithm 3.1.*

Proof. If some correct process p_j does not receive its expected item, it does not send a PROCEED vote. Thus no process obtains the PROCEED vote from p_j. Since no process receives all n PROCEED votes and a clue is composed of a signed set of n PROCEED votes, no process produces its clue. From the *no creation* property of perfect links, no process receives any clue. Without a majority of clues, no process is able to unseal the item at line 36 and hence to release the item at line 37 of Algorithm 3.1. □

Theorem 12 (Validity)**.** *If a correct process p_i releases an item x, then either $x \in M$ and x matches d_i, or x is the abort item φ.*

Proof. A correct process p_i explicitly releases the abort item φ at line 30, and the only other case of release is at line 37. In the latter, p_i releases the item that is stored in variable sealedItem. From Lemma 4, if p_i releases at line 37, it has previously received an item matching d_i and stored it in variable sealedItem. From the *no creation* property of perfect links, that item was sent and offered by some process of Π. Since, from line 13, no subsequently received items can be stored in sealedItem, at line 37, p_i releases an item that belongs to M and matches d_i. □

Theorem 13 (Uniqueness)**.** *No correct process releases more than once.*

Proof. The boolean variable released and the atomic execution assumption prevent any correct process from releasing more than once. □

Theorem 14 (Non-triviality)**.** *If all processes are correct, no process releases the abort item φ.*

Proof. Since all processes are correct, each process sends the correct encrypted item at line 11 as agreed in the terms of the exchange. From the *reliable delivery* property of perfect links, every process p_i receives an item matching description d_i before time $t_1 = t_0 + \Delta_{\text{PL}}$, so every process produces and sends its PROCEED vote at line 17 in a timely fashion. From the IC2 property of BA, no process receives an invalid PROCEED vote. Therefore, no process releases the abort item φ (line 30). □

Theorem 15 (Termination)**.** *Every correct process eventually releases an item.*

Proof. The time-out at line 18 ensures that every correct process starts all n executions of BA at the same time. This implies that, from the existence of a time bound for the *termination* of BA and the IC1 property of BA, there is a

time after which either (a) every correct process receives at least one invalid PROCEED vote and releases the abort item φ or (b) every correct process receives all n valid PROCEED votes. In the latter case, every correct process then produces and sends its i-th clue at line 27. From the *reliable delivery* property of perfect links and the *honest majority* assumption, every correct process receives a majority of clues and then releases at line 37. □

Theorem 16 (Integrity)**.** *No process p_j releases an item m_i, with process p_i correct, if m_i matches description d_k of some correct process p_k, with $p_k \neq p_j$.*

Proof. Only in a single step of Algorithm 3.1, i.e., at line 11, does a correct process p_i transmit its item m_i through the network. Since p_i is correct, p_i encrypts m_i using the key of p'_k in order to send it through the network. Thus no process other than p_i and p_k holds a deciphered version of m_i and, since p_i and p_k are correct, they do not send a deciphered version of m_i to p_j. From the PKI unforgeability assumption, p_j is not capable of obtaining a deciphered version of m_i. Thus p_j does not release m_i. □

Theorem 17 (Fairness)**.** *If any process p_i releases an item m_j matching description d_i, with p_i or p_j correct, then every correct process p_k releases an item matching its description d_k.*

Proof. The proof is by contradiction.
Assume that some correct process p_k does not release an item matching description d_k and that some other process p_i releases an item m_j matching description d_i, with p_i or p_j correct. If p_i releases m_j (line 37), from assumption on SB, p_i must have received a majority of clues in some previous steps: either (1) because p_i is correct or (2) because p_j is correct and encrypted m_j before sending it to p_i (lines 10 and 11). From the *honest majority* assumption, at least one of these clues is produced by some correct process p_x. Process p_x therefore receives all n PROCEED votes. Thus, from the IC1 property of BA, every correct process receives all n PROCEED votes. This implies that no correct process releases the abort item φ (line 30), including p_k. From the *validity* and *termination* properties of FE, if p_k does not release the abort item φ, then p_k releases an item matching its description d_k. A contradiction. □

3.4 Discussion

3.4.1 Are we being unfair to Byzantine processes?

Without gaining any advantage over correct processes, a Byzantine process may systematically have all processes abort the exchange. However, this is

only the case in the early stages of our algorithm, during the transmission of the items or during the voting phase. Once the vote has been successfully completed, correct processes execute the remainder of the protocol without depending on the behavior of any Byzantine process. In other words, correct processes may complete the exchange possibly at the expense of Byzantine processes. So the question is: are we being unfair to Byzantine processes? The answer is twofold.

Consider the case where a Byzantine process is truly malicious, in the sense that, behind the process, there is a malevolent mind purposefully trying to abuse the system. One can argue that the abuser is responsible for his own fate. Even more so, since, if the correct processes are able to obtain their items, there is nothing preventing a malicious process from successfully receiving its item, even if it tried to corrupt the system, as illustrated in Figure 3.4(b).

Where a Byzantine process is merely malfunctioning, without any malicious intention, e.g., in the case of a crash, a failing process might not be responsible for the occurrence of the failure, so unfairness would then arguably prejudice an otherwise correctly behaving process. This problem can be solved by assuming a crash-recovery model, fair-loss (instead of perfect) links [ACT98] and some local persistent storage available to each process.[4] Indeed, with fair-loss links, if the fair exchange protocol successfully reaches the key exchange phase, a *good process*[5] would eventually receive a majority of keys, and thus be able to release its item. On the other hand, the use of fair-loss links does not prevent an unfair outcome for a process that *definitively crashes*, i.e., a *bad process*. However, in such a catastrophic event, one can argue that unfairness is less of an issue, since the digital item is lost anyway.

3.4.2 Complexity Analysis

The performance of our solution is directly dependent on the performance of the underlying modules used in Algorithm 3.1 and in particular on that of the BA module. If BA reaches a decision in s communication steps, Algorithm 3.1 needs $s + 2$ communications steps to reach termination. In [DR82], it was shown that the lower bound for deterministic BA is $b + 1$. This result provides us with a best possible performance of $b + 3$ communications steps for Algorithm 3.1, with b Byzantine processes. Thus, in the worst case scenario, termination is reached in $\lceil \frac{n}{2} + 2 \rceil$ communication steps.

[4] Upon recovery, such a persistent storage allows a crashed process to resume its activity from where it stopped.

[5] A *good process* is a process that can crash and recover many times, but that eventually remains up [ACT98].

Regarding the number of messages, the cost analysis may be done for both a *broadcast* network, in which a broadcast needs one message, and for a *point-to-point* network, in which a broadcast needs $n - 1$ messages. In either cases, the performance of Algorithm 3.1 in terms of number of messages depends on the number of messages σ needed in BA. In a broadcast network, our solution needs $2n + \sigma$ to reach termination. In a point-to-point network, our solution needs $n^2 + \sigma$ to reach termination. Results in [DR82] show that, for a point-to-point network, the lower bound on the number of messages to reach agreement in BA is $O(nb)$. So the message cost for Algorithm 3.1 does not depend on the number of Byzantine processes and our solution to fair exchange in a point-to-point network requires $O(n^2)$ messages to reach termination.

Chapter 4

A Java Implementation

Criticism is something we can avoid easily
by saying nothing, doing nothing,
and being nothing.

Aristotle

Abstract. The main aim of this chapter is to present a pedagogical tool, developed in the context of this thesis, for illustrating and apprehending the complexity of fair exchange. We thus start by proposing a Java implementation of the modular fair exchange algorithm detailed in the previous chapter. The application is completed with a simple graphic user interface in order to set up, run and display executions of our modular protocol. The exercise of implementing such an application leads to two secondary achievements: (1) illustrating the challenges of translating from logical pseudo language to programming language and (2) providing an implementation of Byzantine behaviors. The wide range of Byzantine behaviors is achieved by an approach that consists in refactoring the correct behavior and then creating Byzantine subclasses.

4.1 Introduction

By presenting a practical Java implementation of fair exchange in the context of a pedagogical tool, this chapter proposes a break from theoretical discussion, as illustrated in Figure 4.1. However, it is logically linked to Chapter 3, since the implementation is based on Algorithm 3.1. The aim is to allow for a better understanding of such a protocol by providing a means to set up specific

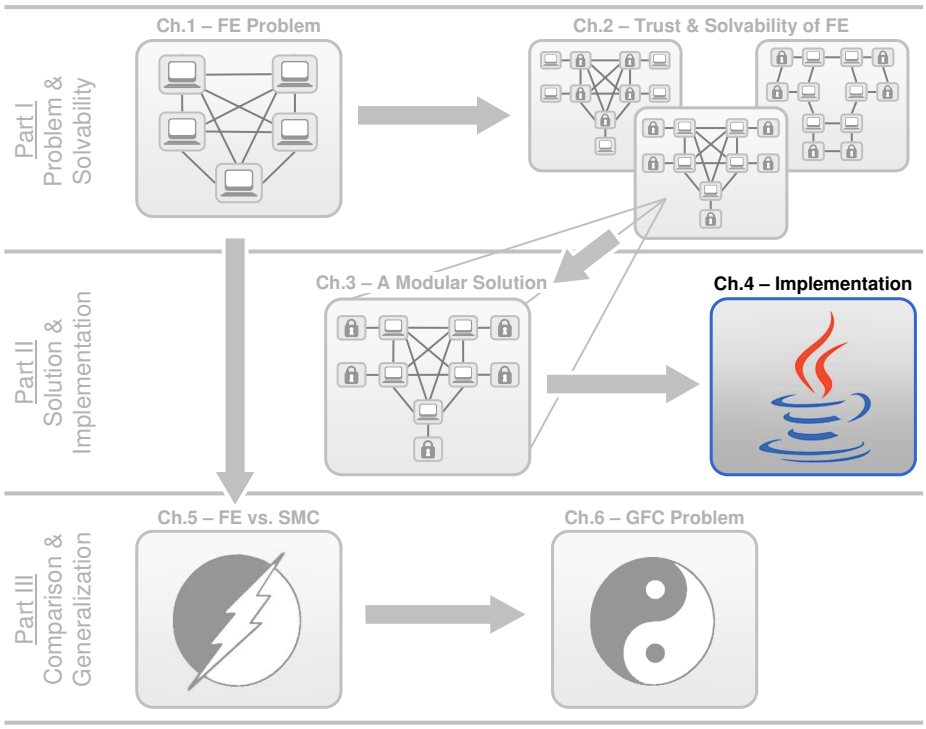


Figure 4.1: Thesis outline.

executions of the protocol and monitor them step-by-step through a graphical user interface. The application is thus composed of the correct implementation of fair exchange, implementations of a wide range of Byzantine behaviors and the graphical user interface.

4.1.1 From a Pseudo to a Real Programming Language

When describing an algorithm, the use of a pseudo programming language is appropriate precisely because the high-level syntax of such a language allows us to focus on the semantics of the algorithm, while hiding irrelevant implementation details. However, certain implicit assumptions inherent to this syntax need to be addressed when implementing an algorithm in a real programming language such as Java.

While certain abstractions, e.g., the *fair exchange* (FE) module, are explicit in the theoretical architecture shown in Figure 3.3 others are implicit. This is the case with the notion of process, which obviously requires a corresponding abstraction in the Java implementation. Figure 4.2 shows the structure of our

implementation, which is based on the theoretical architecture. Each module employed in our solution is represented by one instance of a respective class, e.g., the `SB` class implements the behavior of the *secure box* (SB) module. For simplicity of implementation, the abstract *Byzantine agreement* (BA) module is divided into n instances[1] of the `BA` class, each running an execution for a distinct General, i.e., one instance of the module per voter.[2] The perfect link module is implemented by two classes: `Network` and `NetworkQueue`.

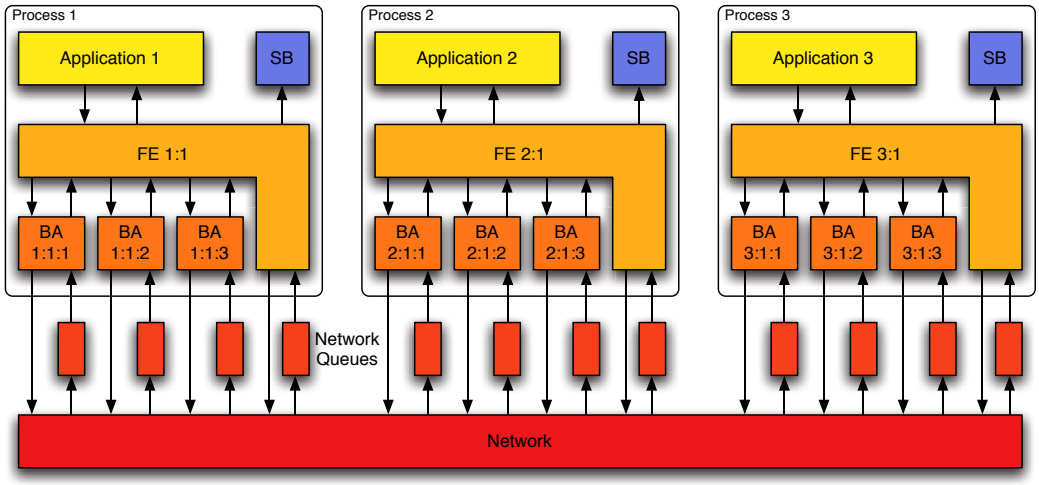


Figure 4.2: Communication architecture.

In a pseudo programming language, as used in Chapter 3, the ability of the various modules to communicate with counterparts is implicit. In a given process, modules communicate with the module above or the ones below them by calling primitives. In order to communicate with their corresponding modules in other processes, they rely on message passing.

In Java, while calling primitives is done using the reference of an object, message passing requires the explicit identification of the modules in the communication architecture in order to deliver properly the various messages, i.e., each module must be identified by a distinct address. In a specific exchange, processes are identified by an integer ranging from 1 to n. An instance of the `FE` class is thus identified by the process and exchange numbers, e.g., "2:1" as in Figure 4.2. The identification of a `BA` object is obtained by the concatenation of the ID of the `FE` object with the process number of its General, e.g., "2:1:3" identifies the `BA` instance running on process 2 that will deliver the vote of process 3 (in exchange 1).

Another major difficulty occulted by pseudo languages is the management

[1]The number n is the total number of processes in an exchange.
[2]In Byzantine agreement, the General is the process broadcasting the message.

of concurrency. In the description of a distributed algorithm, it is implicit that parts of the code need to be executed concurrently. For example, in Algorithm 3.1, the *upon* syntax does not specify how the code will be executed, i.e., by which thread of execution, and how atomicity will be achieved. This particular issue is discussed when presenting the sequence diagram in Section 4.2.2.

4.2 A Java Implementation of Fair Exchange

In this section, we present the implementation of our modular solution, i.e., Algorithm 3.1 of Section 3.3. The core of our application is described using UML[3] [ALH98, FOW03], i.e., through a static model (class diagram) and a dynamic one (sequence diagram). The functional model (use-case diagrams) is omitted since it corresponds to the functional and behavioral requirements of fair exchange, thoroughly described in Chapter 1.

4.2.1 Static Model

The implementation of our modular protocol relies on a series of Java classes, each mapping a single module of the layered architecture. Figure 4.3 provides a complete class diagram of the implementation of Algorithm 3.1. While the `FE`, `BA` and `SB` classes play the respective roles of the modules of our solu-

[3]Unified Modeling Language.

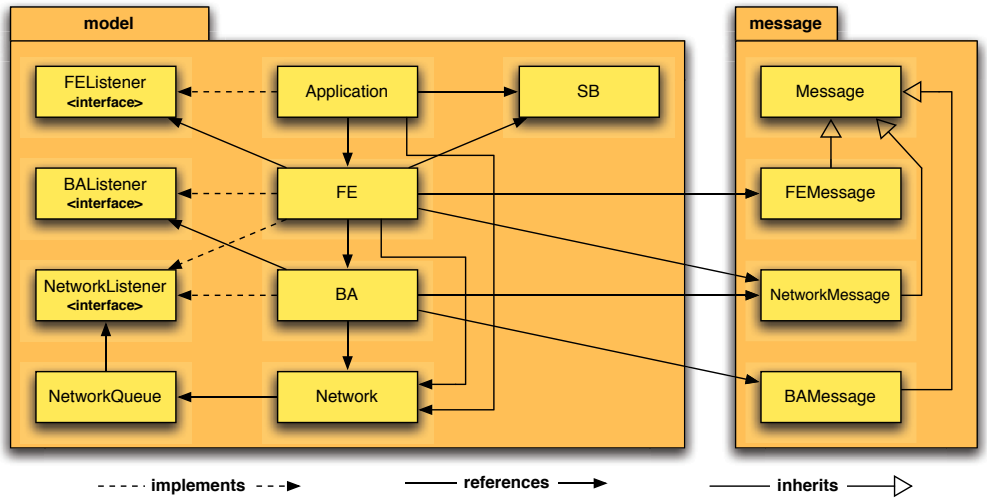


Figure 4.3: Class diagram.

tion, i.e., the *fair exchange*, *Byzantine agreement* and *secure box* modules, the `Network` and `NetworkQueues` classes correspond to the *perfect link* module. In order to communicate with one another, while preserving as little coupling as possible, the classes implement one or more of the following Java interfaces: `FEListener`, `BAListener` and `NetworkListener`.

The *listener* pattern achieves the intended modularity by keeping the lower layered modules unaware of the modules built on top of them. For example, the `FE` module has a reference to the `Network` module but the opposite is not true. So, in order to receive messages from processes in the network, the `FE` module registers to the `Network` module as a `NetworkListener`. Any module implementing the `NetworkListener` interface can thus register to the `Network` module. Moreover, modularity is further achieved through the use of a specific Java interface for each module. These interfaces, not shown in Figure 4.3, allow the implementation and the specification of modules to be kept separate. In the above example with the `FE` and `Network` modules, the `FE` module actually references the `Network` interface, not the class, so the implementation of the network can be modified without impacting the implementation of any other module. Messages transmitted through these interfaces are found in the package `message` and are all subclasses of the super-class `Message`.

4.2.2 Dynamic Model

Figure 4.4 shows a detailed sequence diagram of the execution of the fair exchange protocol, while Figure 4.5 shows the fair exchange algorithm mapped against the execution sequence of the code of the `FE` class taken from Figure 4.4.

In each process, the `Application` thread (Thread 1 in Figure 4.4) initializes an instance of the `FE` class by calling the `constructor` method. The `FE` module connects to the network, which assigns it to an instance of `NetworkQueue`, and creates n instances of the `BA` class. The BA modules connect to the network, which again assigns a respective instance of `NetworkQueue`, and start their own thread of execution (Thread 2). The `NetworkQueue` objects of all modules also start their own threads of execution (Thread 3 for the `NetworkQueue` objects of `FE` modules and Thread 4 for those of BA modules). At this point, the threads of both the `BA` modules and the `NetworkQueue` objects are ready and waiting. Once Thread 1 has created all the modules, it goes into wait mode, using the *wait-and-notify* monitor of an instance of the `Trigger` class, which is used as a synchronizer. The last Thread 1 to finish the creation process triggers the beginning of the exchange by notifying all the others through this same instance of the `Trigger` class, ensuring that all processes start the exchange at roughly the same time.

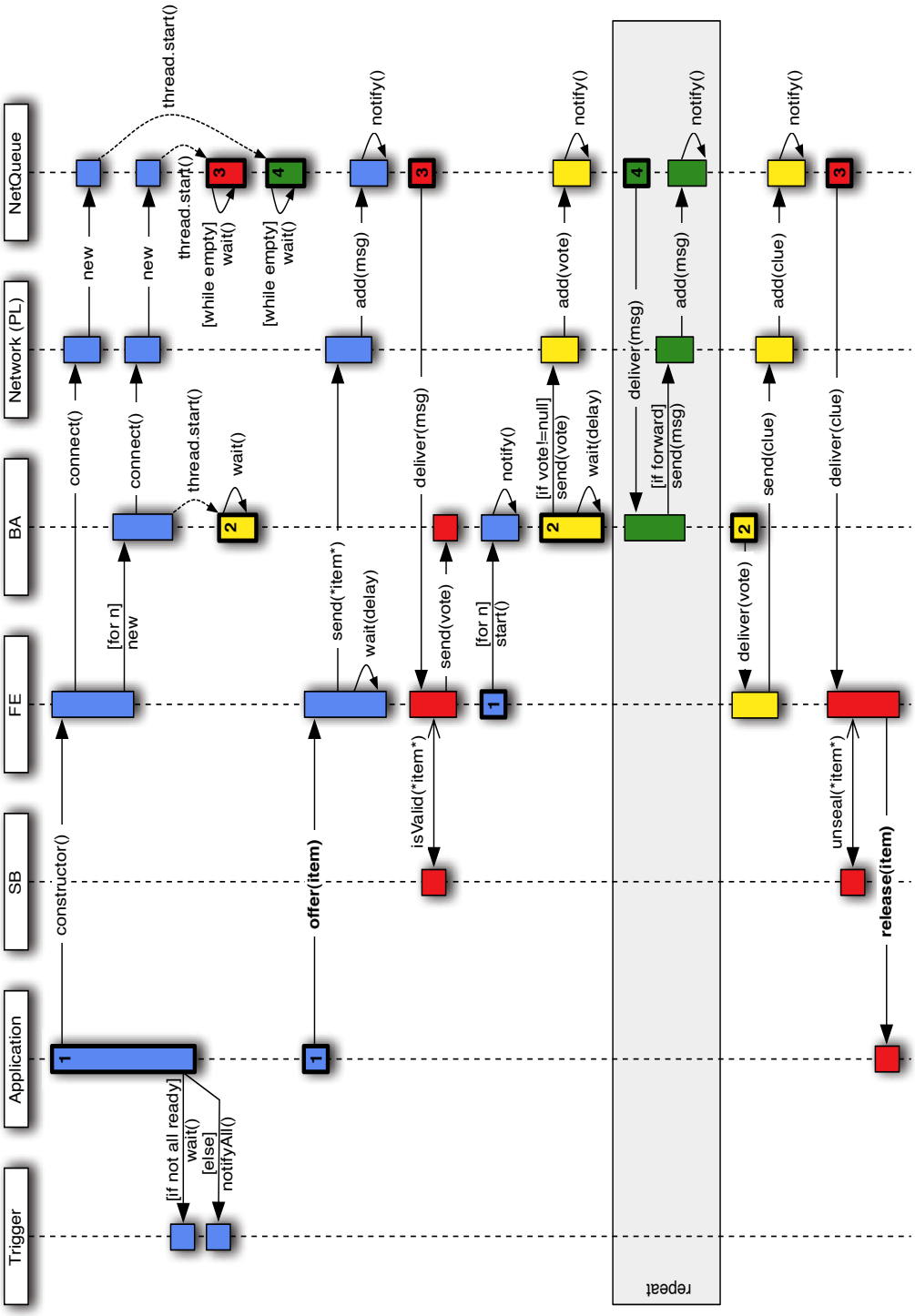
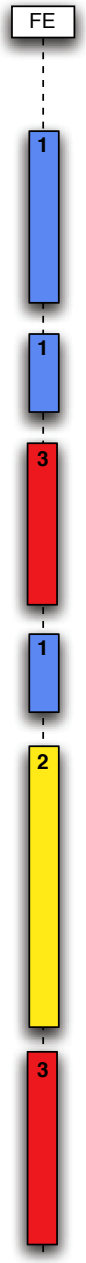


Figure 4.4: Sequence diagram.



Algorithm 3.1 Fair exchange protocol executed by process p_i

```
1:  Uses:
2:      Perfect Link (PL), Byzantine Agreement (BA), Secure Box (SB)

3:  Initialisation:
4:      t_0 ← time()                                   {set t_0 to starting time}
5:      d_i ← ...                                      {set description to known value}
6:      released ← FALSE                               {set variable to false}
7:      sealedItem ← ⊥                                 {set variable to null}
8:      votes, clues ← ∅                               {set variables to empty set}

9:  function offer(m_i, p_r)
10:     item ← encrypt(m_i, p'_r)                      {encrypt m_i using public key of p'_r}
11:     PL.send(p_i, p_r, 'ITEM', item)                {send encrypted item to p_r}

12: upon PL.deliver(p_s, p_i, 'ITEM', item) do         {PL callback}
13:     if (sealedItem = ⊥) then                       {check for duplicate send}
14:         sealedItem ← item                          {store received item}
15:         if SB.isValidItem(sealedItem, d_i) then    {ask SB to check item}
16:             vote ← sign('PROCEED')                 {produce PROCEED vote}
17:             BA.send(p_i, vote)                     {send vote}

18: upon time() > t_0 + Δ_PL do                        {item exchange phase is over}
19:     for all p_j ∈ Π do                             {for all processes}
20:         BA.start(p_j)                              {start BA}

21: upon BA.deliver(p_j, vote) ∧ (¬released) do        {BA callback and not released}
22:     if validProceedVote(vote) then                 {check vote}
23:         votes ← votes ∪ vote                       {add p_j's vote to set}
24:         if (|votes| = n) then                      {if all votes are PROCEED}
25:             for all p_k ∈ Π do
26:                 clue ← sign(votes)                 {produce clue}
27:                 PL.send(p_i, p_k, 'CLUE', clue)    {send clue}
28:     else
29:         released ← TRUE                            {set variable to true}
30:         release(φ)                                 {release φ}

31: upon PL.deliver(p_j, p_i, 'CLUE', clue) ∧ (¬released) do   {PL callback and not released}
32:     if validClue(clue) then                        {check if message is valid}
33:         clues ← clues ∪ {clue}                     {add p_j's clue to set}
34:         if (|clues| > n/2) then                    {check if majority of clues}
35:             released ← TRUE                        {set variable to true}
36:             item ← SB.unseal(sealedItem, clues)    {unseal item using SB}
37:             release(item)                          {release the unsealed item}
```

Figure 4.5: Threads running Algorithm 3.1.

As shown on Figure 4.4, the actual exchange starts by having each Thread 1 call the `offer` method of the `FE` module. The execution then follows precisely our fair exchange algorithm, i.e., Algorithm 3.1. However, it is interesting to notice how the different threads combine to execute the code of the algorithm, as shown in Figure 4.5. Thread 1 runs the *item exchange* phase of Algorithm 3.1, by sending the item to the corresponding `FE` module, and then waits for the duration of a given value of the delay Δ_{PL}. Thread 3 (the `NetworkQueue` thread of the `FE` module) delivers the item, checks with the `SB` module whether the item matches its description and sends the vote to the `BA` module. Once the delay Δ_{PL} is over, Thread 1 triggers the *voting* phase by notifying each Thread 2 (the `BA` thread) running on the same process. The `BA` modules go through their own protocols, shown inside the repeat box and described in the following section, and their threads then deliver the votes of each process. If all votes are gathered correctly, the last Thread 2 to finish starts the *clue exchange* phase by sending the clue to all the processes. Finally, Thread 3 delivers the clues and, once a majority is reached, calls the `unseal` method of the `SB` module and releases the item.

4.2.3 Byzantine Agreement Module

Since our solution relies on a *Byzantine agreement* (BA) module, it requires providing its implementation. In our application, the implementation of the Byzantine agreement protocol is based on the algorithm with signed messages presented in [LSP82]. The protocol provides a means to broadcast messages reliably, which is used to broadcast the votes in the case of fair exchange. Intuitively, this is achieved by having all the processes repeatedly sign and echo messages for a certain number of rounds. For the Byzantine agreement protocol with signed messages, this means going through $b + 1$ rounds, with b being the maximum number of Byzantine processes. Therefore, in the context of our fair exchange solution, where there is a majority of correct processes, the required number of rounds is $\lfloor \frac{n+1}{2} \rfloor$.

Every `FE` module creates n `BA` modules, each having a distinct process as its General, i.e., the process sending its vote. In the first round, the General sends its vote to all the Lieutenants, i.e., all the other processes.[4] The Lieutenants then sign and echo the vote until it reaches a majority of processes, i.e., until the vote has gone through the required number of rounds. If all the messages of a single execution of `BA` are valid PROCEED votes, the `BA` module delivers a PROCEED vote to the `FE` module. Otherwise, if messages are missing or one of the messages contained some other value, it delivers an ABORT vote.

[4] In the analogy used in [LSP82], the terms *General* and *Lieutenants* correspond respectively to the sender and the receivers of a message broadcast.

In [LSP82], the fact that processes are able to detect the absence of a message is an important requirement. For example, in the first round, a Lieutenant should be able to detect that it did not receive the General's message. The assumption was explicitly implemented in our application by calculating the total number of messages that a process should receive in a correct execution of the Byzantine agreement protocol. This implies being able to verify that messages are not received more than once, i.e., messages with the same list of signatures should be taken into account only once. The total number of messages is easily computable from the number of messages a process should receive in a given round, which is a function of n, the number of processes, and r, the round number. In a round r, a Lieutenant p should receive one message per possible arrangement of r process signatures. However, the first signature is necessarily the General's and the message does not include the signature of p. Thus, in each round, the number of messages any Lieutenant should receive is a permutation of $n-2$ elements over $r-1$ positions:

$$P_{r-1}^{n-2} = \frac{(n-2)!}{(n-r-1)!}\,.$$

The total number Q of expected messages for a single Lieutenant can be computed as the sum of the permutations for each rounds. When n is an even number,

$$Q = \sum_{r=1}^{\frac{n}{2}} \frac{(n-2)!}{(n-r-1)!}\,,$$

and when n is odd,

$$Q = \sum_{r=1}^{\frac{n+1}{2}} \frac{(n-2)!}{(n-r-1)!}\,.$$

At the end of a BA execution, the number of messages received is thus verified against the number of expected messages. Table 4.1 presents the values for the number of expected messages for various numbers of participating processes.

4.2.4 Secure Box Module

As argued in Chapter 3, the aim of our solution is to rely on the *secure box* (SB) module as little as possible. Its implementation is thus reduced to two simple methods: one for verifying that the item received matches its description and the other for unsealing the item once the proof has been gathered. For simplicity and performance, the current implementation only simulates the use of

Table 4.1: Number of messages per process in a given round of BA.

	Number of processes								
Round	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1
2	-	1	2	3	4	5	6	7	8
3	-	-	-	6	12	20	30	42	56
4	-	-	-	-	-	60	120	210	336
5	-	-	-	-	-	-	-	840	1680
Total	1	2	3	10	17	86	157	1100	2081

encryption by adding a special value to the item. At this point, proper encryption using a PKI infrastructure is not necessary since we essentially want to build a pedagogical tool. However, due to the modularity of our solution, replacing the `SB` module by one relying on a real PKI infrastructure is quite straightforward.

4.2.5 Network Module

The perfect link module of our solution is implemented by relying on the `Network` and `NetwordQueue` classes. The latter is a basic message queue with a dedicated thread that delivers messages following a FIFO[5] order. The former provides a means for processes to send messages reliably through the network. Since the purpose of the application is to illustrate executions of the protocol, for simplicity, the implementation is local, i.e., it relies on local references instead of remote ones, as provided by the RMI[6] infrastructure, or concrete network addresses. Again, however, the modularity of our structure would allow the implementation of the `Network` class to be modified to support distributed executions. This particular implementation thus implies that all the processes run on the same virtual machine.

[5]First In, First Out.
[6]Remote Method Invocation.

4.3 Implementing Byzantine Behaviors

When illustrating executions of fair exchange or any other Byzantine-failure-tolerant algorithm, a key aspect is the need to implement Byzantine behaviors. Exhibiting a fault-tolerant protocol without the occurrence of failures is rather meaningless. However, implementing a wide range of Byzantine behaviors can be challenging. We therefore propose to address this task efficiently by refactoring the implementation of the correct behavior and then building a hierarchy of Byzantine behaviors all inheriting from the correct behavior. While this is carried out in the context of our fair exchange protocol, the interesting aspect of this contribution is that it is a general approach applicable to any Byzantine-tolerant algorithm.

4.3.1 Refactoring the Correct Implementation

Our approach consists in taking advantage of inheritance to allow Byzantine implementations to modify any single line of code of the correct protocol. The first step is thus to refactor the structure of the correct implementation. For example, Code Listing 4.1 shows the correct implementation of an excerpt of Algorithm 3.1 before modification (see Figure 4.5). Lines 12-17 of Algorithm 3.1 are mapped to lines 3-6 of Code Listing 4.1.

Code Listing 4.1: Java method before modification.

```
1  public void deliver(NetworkMessage message){
2    if (!released)
3      if (message.getType().equals("item") && sealedItem == null)
4        if(sb.isValidItem(message.getObject(),description)){
5          sealedItem = message.getObject();
6          baGeneral.send(sign("PROCEED"));
7        }
8      else if (message.getType().equals("clue"))
9          [...]
10 }
```

The refactoring is achieved by first splitting the code into sub-methods, each containing a single line of the algorithm. In Code Listing 4.2, the lines 5 and 6 of Code Listing 4.1 are respectively located at lines 18 and 23. We then prepare the correct implementation in order to support crash-stop and crash-recovery behaviors by adding verifications of the status of the module before executing any single line, e.g., lines 16 and 17. In the case of crash-recovery, the method `crashRecovery` is implemented in order to put in wait

Code Listing 4.2: Java method after modification.

```
1  public void deliver(NetworkMessage message){
2    crashRecovery();
3    if (!released && !crashed)
4      if (message.getType().equals("item") && sealedItem == null)
5        deliverCryptedItem(message);
6      else if (message.getType().equals("clue"))
7        [...]
8  }
9  protected void deliverCryptedItem(NetworkMessage message){
10   if(sb.isValidItem(message.getObject(), description)){
11      deliverCryptedItem01(message);
12      deliverCryptedItem02(message);
13   }
14 }
15 protected void deliverCryptedItem01(NetworkMessage message){
16   crashRecovery();
17   if(!crashed)
18     sealedItem = message.getObject();
19 }
20 protected void deliverCryptedItem02(NetworkMessage message){
21   crashRecovery();
22   if(!crashed)
23     baGeneral.send(sign("PROCEED"));
24 }
```

Code Listing 4.3: Implementing a method to allow crash-recovery behaviors.

```
1  protected void crashRecovery(){
2    while(down)
3      synchronized (this)
4        try
5          this.wait();
6        catch (InterruptedException e)
7          e.printStackTrace();
8  }
```

mode any thread trying to run the code of a module that is set to `down` (see Code Listing 4.3). Finally, a dedicated crash-recovery thread is added in the correct implementation. However, it stops immediately, i.e., the `run` method is left empty, and is only added in order to be overridden in Byzantine subclasses.

By creating sub-methods for each line, the modifications obviously add some

complexity to the code of the correct implementation. However, the refactoring allows for better re-use of the code of the correct behavior, by permitting each line of the algorithm to be isolated in order to achieve deviating behaviors, while leaving the rest of the code unchanged. This provides flexibility for implementing any type of Byzantine behavior: crash-stop, crash-recovery or malicious failures.

4.3.2 From Correct to Byzantine through Inheritance

The correct implementation is used as a superclass for creating Byzantine subclasses, as shown in Fig 4.6. By adequately overriding just one method of the superclass, i.e., a single line in the correct algorithm, a Byzantine implementation can be produced. For example, producing a crash-stop behavior only requires overriding one of the methods of the superclass with the method of Code Listing 4.4. The example is given for the correct `deliverCryptedItem02` method presented in Code Listing 4.2.

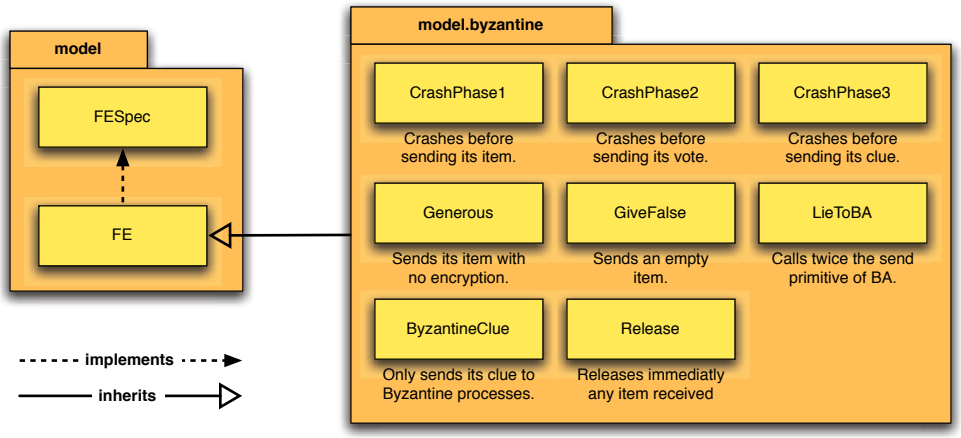


Figure 4.6: Class diagram of the Byzantine behaviors.

Code Listing 4.4: Overriding a method to produce a crash-stop behavior.

```
1 protected void deliverCryptedItem01(NetworkMessage message){
2   crashed = true;
3 }
```

For producing a crash-recovery failure, the `run` method can be implemented as in Code Listing 4.5. In the example, while the status of the module is not set to either `good`[7] or `crashed`, the *crash-recovery* thread regularly modifies the value

[7]Captures the status of a crashing-and-recovering module that has eventually stopped

of the boolean variable `down`. The delay of the loop can range from constant to totally random, as well as decreasing or increasing, by implementing the `getSleepDelay` method accordingly. The `eventualStatus` method provides a means to implement either *good* or *bad* modules, i.e., according to the desired behavior this method can set the status of the module to `good` or `crashed`, in which case the crash-recovery behavior stops.

Code Listing 4.5: Run method for the crash-recovery behavior.

```
1  public void run(){
2    while(!good && !crashed){
3      try
4        Thread.sleep(getSleepDelay());
5      catch ( InterruptedException e )
6        e.printStackTrace();
7      // Allows to eventually set status to good or crashed
8      eventualStatus();
9      if(down || good || crashed){
10       down = false;
11       synchronized (this)
12         this.notifyAll();
13     } else
14       down = true;
15   }
16 }
```

The crash-recovery behavior requires that threads executing the protocol can be (1) put on hold as soon as the module is *down* and (2) released when it is *up* again. The latter is achieved at line 12 of Code Listing 4.5 by having the *crash-recovery* thread notify all the threads that were waiting since the module was `down`. Once the `notifyAll` method is called by the *crash-recovery* thread, executing threads are able to proceed with the normal execution of the protocol, unless the status has meanwhile been set to down again.

Implementing truly malicious behaviors may require overriding methods with more complex code. However, the inheritance from the superclass is convenient, since it provides an otherwise correctly behaving malicious module. More sophisticated malicious behaviors may be achieved by combining Byzantine behaviors through a hierarchy of Byzantine subclasses.

crashing. Note that, in the crash-recovery failure model, the term *good* designates both correct and eventually correct processes.

4.4 A Visualization Tool

One of the goals of the implementation is to provide a tool, i.e., a graphic user interface, allowing the execution of our fair exchange protocol to be monitored. The structure of our application follows the Model-View-Controller (MVC) architectural pattern [Bur92]. It allows the isolation of the data (model) from user interface (view) and data access (controller) concerns, so that changes to the user interface do not affect the data handling. In our application, the model corresponds to the implementation of the fair exchange protocol presented in Section 4.2, while the graphic user interface provides both the view and the controller. This is illustrated in Figure 4.7, which extends the class diagram of Figure 4.3. The `view` package contains the classes of the graphic user interface presented below. The `support` package gathers the classes required by most others, such as the classes `Preferences` and `LogWriter`, which allows objects to write into the log file during executions of the protocol.

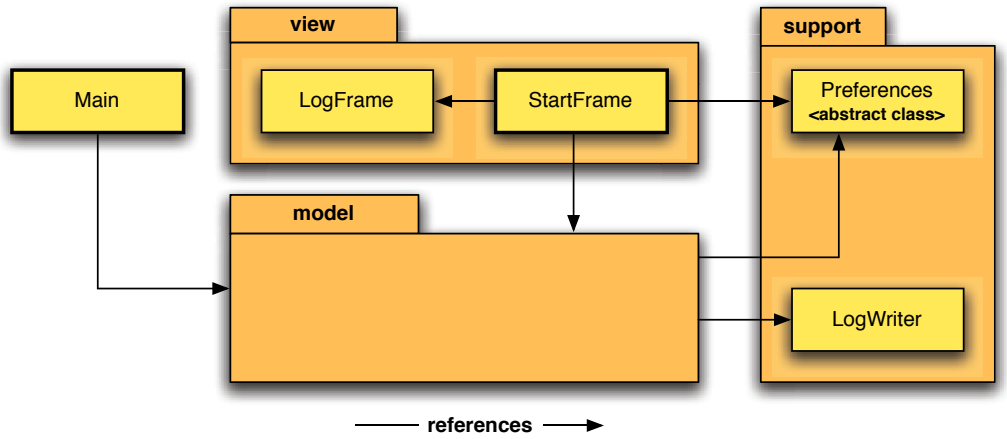


Figure 4.7: Extended class diagram.

The difficulty of providing a practical visualization tool resides in observing executions without affecting their normal outcome. While direct observation of an execution may be possible, it is mostly impractical since executions cannot be stopped or even slowed down without impacting them. Our solution thus consists in relying on the log of the execution, which can then be viewed at any convenient speed. The visualization tool combines two windows, one for setting up and triggering an execution, and the other for displaying the resulting log file of the execution.

4.4.1 Execution Setup

Figure 4.8 shows the setup window of our application, where the preferences for an execution can be configured. From this window, a user can choose to create a log file for a new execution or open a file previously stored on the hard drive.

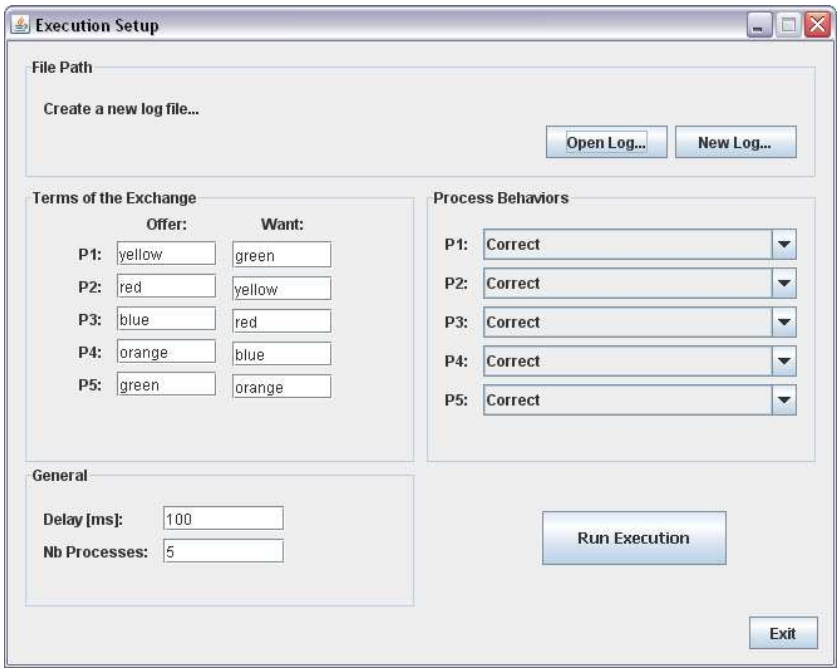


Figure 4.8: Setup window.

To create a new log file, the user must first define a path for the file by using the New Log button. He can then modify the default terms of the exchange, which are defined by the `string` values in the *offer* and *want* textfields. A specific behavior for each participant process can also be chosen. There is no limitation on the number of Byzantine processes, since illustrating bad executions may be interesting for pedagogical purposes. Finally, the user can set the delay for a single transmission on the network and the number of processes in the exchange. The delay corresponds to the value of Δ_{PL} as defined in the previous chapters. Once all the preferences have been defined, the user can trigger the execution of the exchange by using the Run Execution button, which then opens the monitoring window. While our graphic user interface allows a maximum of five processes, an exchange with any number of processes can be executed by using the command line and the class `Main`, as shown in Figure 4.7. This last feature is a direct result of the adoption of the MVC architectural pattern, since the model is completely isolated and independent of any choices made regarding the user interface.

Alternatively, the user can simply click on the Open Log button and choose an existing log file. Once a file is selected, the monitoring window opens to display the details of the logged execution.

4.4.2 Running an Execution

Figure 4.9 shows the two windows composing our graphic user interface, corresponding to the two classes of the `view` package, and the steps allowing executions to be visualized. Once the Run Execution button is pressed, the preferences are updated and the execution of the exchange starts. The preferences are then retrieved by the model and the execution follows the description of Figure 4.4. Just before the execution of the fair exchange protocol, a log file is created. This is a text file written into during the execution of the exchange in order to log the actions of each process.

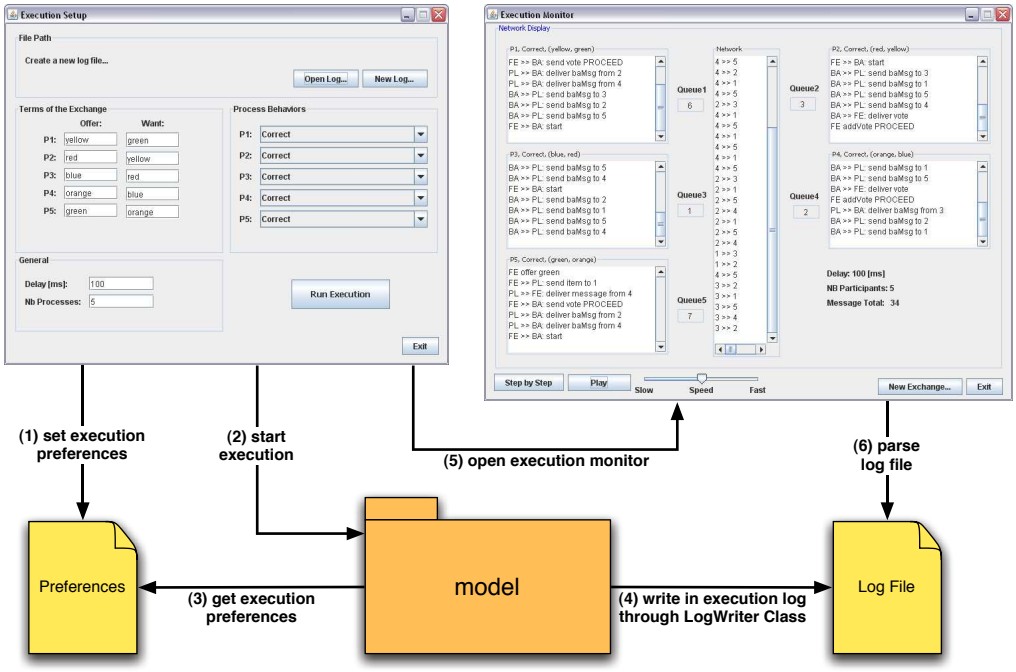


Figure 4.9: Execution sequence of the graphic user interface.

After the exchange protocol is completed, the second window appears. This window retrieves the log file and parses the first lines of the text file to display the labels according to the preferences. The parsing of the rest of the file is then done according to the actions performed by the user through the interface of the execution monitor.

4.4.3 Execution Monitor

The monitoring window of our application shown in Figure 4.10 is composed of six textfields, one for the network and five for the processes. The textfields display the actions taken by the respective processes. Smaller boxes link the network textfield with the process textfields by displaying the number of messages waiting in the incoming message queue of each process. The labels of each textfield and those found in the lower right of the window recall the preferences chosen in the setup window. The bottom of the window provides the control buttons for viewing the execution of the exchange. An execution can thus be displayed at any speed. It can be paused momentarily with the Play/Pause button, slowed down or speeded up using the Speed slider and also viewed action by action with the Step by Step button. The New Exchange button opens the setup window for a new execution.

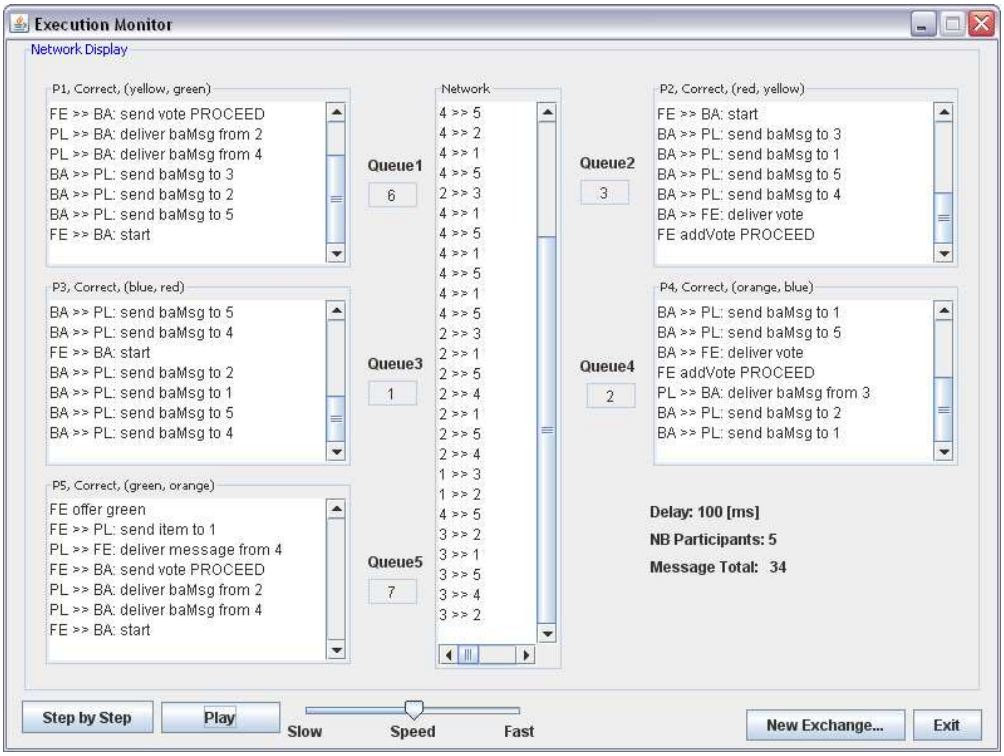


Figure 4.10: Monitoring window.

While this display is an interesting pedagogical tool, a potential improvement would be to make the visualization more graphical, e.g., by having moving graphical elements representing transfers of messages.

Part III

Comparison & Generalization

Chapter 5

Fair Exchange vs. Secure Multiparty Computation

Our shortcomings are the eyes with which we see the ideal.

Friedrich Nietzsche

Abstract. In this chapter, we propose a comparison of fair exchange and secure multiparty computation. Despite their apparent similarity, these two problems arise respectively from the fields of distributed systems and of modern cryptography. The wide differences of description and approach in these research fields render hazardous a straightforward comparison of the various results and solutions. By introducing a common specification framework for the two problems, we examine the differences regarding their generality and properties, and conclude with a discussion on the possible origins of the confusion surrounding certain results found in the literature.

5.1 Introduction

The problem of *secure multiparty computation* (SMC) comes from the field of modern cryptography and consists in allowing a group of parties to compute a specific function securely, i.e., with the input of each party remaining private to all others. The motivation behind this problem is to study to what extent a group of mutually distrustful parties can emulate a centralized trusted third party. Accordingly, the SMC problem is set in a *real* model, with no trusted third party, and is specified with respect to an *ideal* model, in which all the computation is executed by a centralized trusted third party.

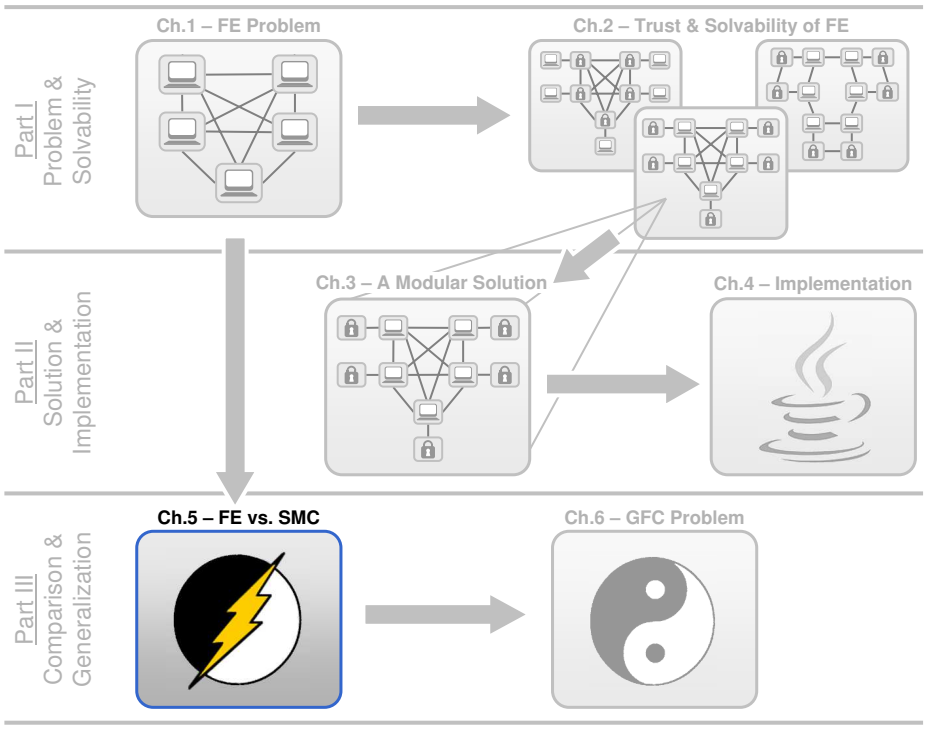


Figure 5.1: Thesis outline.

The problem of *fair exchange* (FE) on the other hand comes from the general problem of exchanging values between mutually distrustful parties. As presented in the properties of Section 1.3.2 (page 16), the exchange is fair if either all parties obtain their desired values or no party obtains anything useful. The approach of fair exchange is opposed to that of secure multiparty computation, as illustrated in Figure 5.1, in the sense that its problem specification is not based on any ideal model. In other words, in the case of fair exchange, the centralized trusted third party is merely a solution to the problem and not a standard of measurement.

This chapter proposes a comparison of these two problems, which are confusingly similar. On one hand, the close connection that both problems have with the concept of trusted third party may indeed give the impression that they are somehow equivalent. On the other hand, several results found in their respective literature, i.e., modern cryptography and distributed systems, seem to be contradictory. Because the specifications of both problems differ greatly in form, making sense out of these confounding results is not straightforward. Moreover one can find very little in the literature about measuring one problem against the other. To compare SMC and FE, our approach consists in first

introducing a common specification framework for describing the problems and then exposing their connections in order to bridge the gap between them.

5.1.1 Secure Multiparty Computation (SMC): a Brief History

The SMC problem was first introduced by [Yao82] and has been extensively studied since then. If the functional definition of the problem is usually described using a mathematical function, the way of defining the security properties or behavioral constraints has evolved considerably from [Yao82]. The definition of the behavioral constraints was first presented through descriptions of properties and later by relying on the *simulation* approach [Can00, Gol04, GL02], in which the level of security in the real model is defined by emulation of an ideal model relying on a trusted third party (TTP). This approach is now widely used.

Whether described by a set of properties or by the simulation approach, the level of security has been of much debate in the literature. While there is a consensus on including the notions of *privacy*,[1] requiring that inputs remain hidden to all other processes, and *correctness*,[2] ensuring that outputs are computed correctly, it is not the case for *fairness* and *termination*. The list below provides an excerpt of the hierarchy of SMC definitions of [GL02], which presents various degrees in the notions of *fairness* and *termination*. As for *correctness* and *privacy*, they are ensured in all definitions.

- *Secure computation with abort* – Ensures *privacy* of inputs and *correctness* of outputs. However, it does not ensure *termination*, i.e., honest parties have no guarantees of receiving an output, so neither is *fairness*.
- *Secure computation with unanimous abort* – Besides *privacy*, *correctness*, ensures that either all honest parties receive their correct outputs or all honest parties abort. In this case, different levels of *fairness* can be considered, depending on what the adversary[3] obtains:
  - *No fairness* – Ensures nothing more than *fairness* among honest parties: the adversary can always take advantage of honest parties.
  - *Complete fairness* – Ensures that honest parties are guaranteed to receive a correct output if the adversary does. Note that, in the

[1] Corresponds to *integrity* in the set of properties of our definition of fair exchange.

[2] Corresponds to *validity* in the set of properties of our definition of fair exchange.

[3] In cryptography, the term *adversary* usually designates the entity controlling all the dishonest parties.

literature, the definition of SMC with complete fairness is usually referred to as *fair computation*.

Although there is a wide range of SMC definitions, it now seems to be standard that the focus is on providing *privacy* and *correctness* but not *termination* or *fairness* [GL02, MR91], i.e., *secure computation with abort*. Indeed, based on [LLR02] the authors of [GL02] argue that, since Byzantine agreement cannot be composed[4] with two thirds or more Byzantine processes, *security* should be decoupled from *agreement*, which is closely related to *termination* and *fairness*. Based on this last argument, the definition of SMC that we consider in this chapter corresponds to *secure computation with abort*.

Not surprisingly, research on SMC has also produced an impressive body of work regarding solutions, impossibilities and lower bounds. In [BGW88, CCD88, RB89] for instance, it is sequentially shown that any multiparty protocol can be achieved in an unconditionally secure manner, provided that the system is synchronous and that there is an honest majority of peers. These results even provide some level of fairness, as also does [GL90]. In the hierarchy of [GL02] mentioned above, the level of fairness in [RB89] matches the definition of *secure computation with unanimous abort and complete fairness*, also known as *fair computation*. The authors of [BGK04] propose an efficient solution to fair computation that relies on security modules inspired by the approach introduced for solving fair exchange [AV03]. However, as argued in Section 5.5, this definition of *fairness* differs notably from the one specified in the context of fair exchange.

5.2 An Integrated Specification Framework

We consider a distributed system consisting of a set Π of n fully interconnected processes, $\Pi = \{p_1, \ldots, p_n\}$. Processes communicate by message passing, as illustrated in Figure 5.2. The system is *synchronous*: it exhibits *synchronous computation* and *synchronous communication*, i.e., there exist upper bounds on processing and communication delays. The *execution* of algorithm A is then defined as a sequence of steps executed by processes of Π. In each step, a process has the opportunity of atomically performing all of the following actions: (1) send a message, (2) receive a message and (3) update its local state. In each step, the process can of course choose to skip any of these actions, e.g., if it has nothing to send. Based on this definition, a *Byzantine process* is one that deviates from A in any way, so a Byzantine process is

[4]The term *composed* relates to the fact of running concurrent executions of a specific protocol. In the case of Byzantine agreement with signed messages, the argument is that messages from one execution could be re-used in order to lie in some other execution.

Byzantine against a specific algorithm A. Byzantine failures can indeed only be defined with respect to some algorithm [DGG05].

The fields of modern cryptography and distributed systems apply their own distinct approaches when describing the problems of SMC and FE, respectively. However both problems are comparable to that of a cooperative multi-player game. The specifications are divided into two distinct parts: (1) a *functional definition*, which is equivalent to the goal of the game, and (2) *behavioral constraints*, corresponding to the rules under which the game is played.

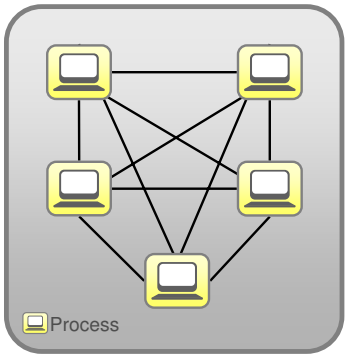


Figure 5.2: A fully connected topology, with five processes.

5.2.1 Functional Definition

Modern cryptography describes the functional definition using a mathematical function, whereas distributed systems rely on a set of primitives. While these approaches may appear different, they are quite similar, with the former being slightly more formal. In modern cryptography, the functionality is described as a whole, with all the inputs and outputs, whereas in distributed systems, it is usually described using one primitive to allow each process to provide its input and a second primitive to allow each process to receive its output value. For example, modern cryptography would define a function $f\colon (y_1, \ldots, y_i, \ldots, y_n) = f(x_1, \ldots, x_i, \ldots, x_n)$. In distributed systems, this function would be translated into the two following primitives along with a description of their semantics:

input(x_i) – Enables process p_i to provide its input x_i.

output(y_i) – Allows process p_i to obtain output y_i. (Works as a callback.)

For the functional definition, the use of a mathematical function is probably preferable, since it is more precise than the description of primitives.

5.2.2 Behavioral Constraints

For the second part of the specification, modern cryptography usually defines the constraints regarding acceptable and unacceptable behaviors by analogy with a model relying on a *trusted third party* (TTP), named the *ideal* model [Gol04, GL02]. Thus the constraints are described by saying that the *real* model, i.e., one not relying on a TTP, should ensure the same properties as in the *ideal* model. In contrast, the field of distributed systems relies on an exhaustive set of properties describing both what should happen and what cannot happen, named respectively liveness and safety properties [GR06d]. Commonly found properties include, e.g., *termination*, which is a liveness property stating that all processes eventually obtain an output value, and *uniqueness*, which is a safety property stating that no process can obtain more than one output value. If the simplicity of the analogy with the *ideal* model seems appealing, we argue that it is inadequate and possibly misleading. This is indeed the case because the guarantees offered by a TTP are too strong and are thus impossible to emulate in the *real* model. The only possible way for the *real* model to live up to the *ideal* model is to downgrade the *ideal* model to a *not-so-ideal* model [Gol04]. Thus, although relying on a set of properties may be less intuitive, it is nonetheless a preferable approach, especially when comparing SMC and FE.

Table 5.1: Specifications in modern cryptography and distributed systems.

	Modern Cryptography	Distributed Systems
Functional Definition	**By defining a mathematical function.**	By declaring a set of primitives.
Behavioral Constraints	By analogy with a trusted third party.	**By declaring a set of properties.**

As summarized in Table 5.1, our specification framework is based on what we believe to be the preferable approach from both worlds, i.e., a mathematical function to describe the functional definition and a set of properties to define the behavioral constraints. In the following, we propose to revisit the specification of SMC and FE using this integrated specification framework.

5.3 Revisiting the Specification of the SMC Problem

The secure multiparty computation problem consists in a group of processes trying to compute a common function securely with the inputs from all processes. The difficulty of the problem resides in achieving *privacy*, i.e., keeping each input hidden from all other processes. Note that Byzantine processes may omit to provide an input value, in which case the computation may not terminate.

5.3.1 Functional Definition

More formally, each process is required to compute securely the result of a well-known deterministic function: $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$, where x_i and y_i are respectively the inputs and outputs of process p_i. Interestingly, in the SMC literature, the domain of definition is usually not clearly defined. This is probably because it is not really an issue when dealing with *correctness* and *privacy*. However, it is essential when *fairness* is at stake.

5.3.2 Behavioral Constraints

Secure multiparty computation allows processes to compute the result of a specific function securely, with each process providing an input value and expecting to receive an output value as the result of the computation. In order to achieve this cooperative goal, there are a number of behavioral constraints to respect, which are described by the set of properties below. At the end of the computation of the function f, we say that process p_i *outputs* a value, meaning that the function returns the output value to p_i. This convention comes from the field of distributed systems and is similar to the one used for classical *deliver* primitives, e.g., in reliable broadcast primitives [HT93].

The SMC problem is usually defined in the modern cryptographic literature by relying on a TTP [Gol04], since the problem was originally described to provide the same guarantees as a TTP. However, in the perspective of bridging the gap between SMC and FE, we define the semantics of SMC using a set of abstract properties. We aim at matching as closely as possible the definition given in [Gol04], which corresponds to the commonly accepted definition of SMC in the field of modern cryptography.

Validity. If a correct process p_i outputs a value y_i, then y_i was computed using function f and with at least the inputs of all correct processes.[5]

Uniqueness. No correct process outputs more than one value.

Non-triviality. If all processes are correct, every process outputs a value.

Privacy. No process p_j outputs the input value x_i or output value y_i of any correct process p_i, apart from what is possibly given away by inputs and outputs of Byzantine processes.

The *validity* and *privacy* properties are intrinsic to the specification of the SMC problem, while *uniqueness* implicitly derives from the ideal model based on a TTP. The *non-triviality* property can be found in most – possibly in all – distributed systems problems, as it becomes evidently necessary when trying to provide any meaningful solution. According to [GOL04], ensuring a liveness property, e.g., *termination*, is not part of the specification of the SMC problem. This is not the case in [BGK04], where the authors rely on secure modules and are thus able to ensure *termination*. However, we argue that [BGK04] describes a stronger problem than the usual problem of SMC and thus, accordingly, omitting *termination* from our specification better captures the essence of SMC as commonly defined in modern cryptography.

5.4 Revisiting the Specification of the Fair Exchange (FE) Problem

The fair exchange problem consists in a group of processes trying to exchange inputs specified ex ante, i.e., each process provides a specific input and expects the input of some other process in exchange. The difficulty of the problem resides in achieving *fairness*. Intuitively, fairness means that, if one process obtains its desired output, then all processes involved in the exchange should also obtain their desired output.

5.4.1 Functional Definition

More formally, each process is required to compute the result of a deterministic function F: $(y_1, \ldots, y_n) = F(x_1, \ldots, x_n)$, where x_i and y_i are respectively the inputs and outputs of process p_i. When all processes are correct, the outcome of F is defined by a function f, a permutation with no fixed points, with

[5] In the literature of SMC, *validity* is usually referred to as *correctness*.

input domain $X = X_1 \times \ldots \times X_n$, where $X_i = \{\overline{x}_i\}$, a set containing a single specific value, and output domain $Y = Y_1 \times \ldots \times Y_n$, where $Y_i = \{\overline{y}_i\}$ and $(\overline{y}_1, \ldots, \overline{y}_n) = f(\overline{x}_1, \ldots, \overline{x}_n)$.[6] Thus the terms of the exchange are defined by f, X and Y. Function f therefore provides the outputs of F when the inputs provided are those expected, i.e., they belong to the domain of f, and the computation achieves completion.[7] However, if this is not the case, F outputs a special value φ for all processes. This special value φ indicates that the exchange aborted. Function F is defined as follows:

$$F(x_1, \ldots, x_n) = \begin{cases} f(x_1, \ldots, x_n), & \text{if } \forall x_i,\ x_i \in X_i \text{ and } f \text{ achieves completion;} \\ \forall y_i, y_i = \varphi, & \text{otherwise.} \end{cases}$$

5.4.2 Behavioral Constraints

Fair exchange allows processes to exchange input values in a fair manner. Each process inputs a value in exchange for a counterpart, of which it can verify the validity. The computation is concluded when every process outputs either a value y_i, which is by definition the result of function f, or the abort value φ, meaning that the computation aborted. The behavioral constraints of the problem of fair exchange are described by the following set of properties.

Validity. If any correct process p_i outputs a value y_i, then either $y_i \in Y_i$ or $y_i = \varphi$.

Uniqueness. No correct process outputs more than one value.

Non-triviality. If all processes are correct, no process outputs the abort value φ.

Termination. Every correct process *eventually* outputs a value.

Privacy. No process p_j outputs the input value x_i or the output value y_i of any correct process p_i, apart from what is possibly given away by inputs and outputs of Byzantine processes.

Fairness. If any process p_i outputs a value y_i, with $y_i \in Y_i$, then every correct process p_j outputs a value y_j, with $y_j \in Y_j$, unless p_i is Byzantine and y_i is computable from the inputs of Byzantine processes.

[6] Since f is a permutation, $\bigcup_{i=1}^{n} X_i = \bigcup_{i=1}^{n} Y_i$.

[7] Byzantine behaviors may prevent the completion of the computation of f.

This set of properties corresponds precisely to the six properties of the specification of fair exchange presented in Section 1.3.2 (page 16). The descriptions of these properties were translated into the new formalism and the *integrity* property was renamed *privacy* in order to match the notion used in SMC.

The *privacy* and *fairness* properties are specific to the problem of fair exchange, i.e., they capture the essence of the problem. However, as discussed in Section 1.3.2, other specifications of fair exchange usually rely on a single property to capture these two notions [ASW00, AGGV05, PG99]. Our set of six fine-grained properties describing fair exchange was first introduced for clarity reasons, to capture the problem better and to facilitate reasoning about the different results and solutions. The choice of two properties (*privacy* and *fairness*), instead of just one, happens to be particularly relevant when comparing SMC and FE, since SMC ensures only *privacy*.

5.5 Comparing SMC and FE

With both problems described using our integrated specification framework, differences that were somehow hidden by implicit assumptions or merely because of the heterogeneous specifications, now become apparent. In this section, we take a first look at the gap between SMC and FE. The differences between the two problems obviously depend on the choices we made regarding what to include in their definitions, in particular for SMC, whose exact definition is still up for debate. Nonetheless, the comparison provides an interesting clarification of the relationship between SMC and FE. Are they equivalent? Is one included in the other? Or do they just have a common intersection? In the following, we answer these questions by pointing out that while SMC is a more general problem, FE is stronger.

5.5.1 The Respective Strengths of SMC and FE

The first major difference is found in the generality of the function computed in both problems. In SMC, the specification allows any function f to be considered, whereas fair exchange only deals with permutations with no fixed points and specific constraints on the inputs of all processes, as described by the terms of the exchange. Secure multiparty computation is far more general than fair exchange with respect to the range of functions it considers. In other words, they are from different levels of abstraction.

The second explicit difference lies in the lists of properties. Fair exchange ensures two properties, *termination* and *fairness*, not ensured by secure mul-

tiparty computation. While these properties can be found in some particular specifications of SMC, they do not belong to the specification commonly used in the modern cryptography literature. Indeed, from [Gol04], we learn that a peer cannot be prevented from abruptly interrupting any execution of SMC at any time, which denies the possibility of ensuring *termination*. Since ensuring true fairness obviously relies on the termination of any execution, this aspect also impacts the possibility of ensuring *fairness*.

In fact, *fairness* is usually not considered in SMC, even though this might seem the case from the emulation approach of a TTP-based ideal model. So not only is *fairness* not ensured, it is also seldom discussed, hence the confusion when comparing seemingly conflicting results in SMC and FE. Moreover, when *fairness* is indeed mentioned in the context of SMC, it usually has a weaker connotation. This last aspect is further discussed in the Section 5.6.

5.5.2 Towards Generalized Fair Computation

To summarize, SMC achieves more in terms of generality, while FE provides two additional properties, i.e., *termination* and *fairness*. This leads us to introduce a middle man in the form of a third problem, which provides both the generality of secure multiparty computation and the set of properties of fair exchange. This new problem, named *generalized fair computation* (GFC), is the subject of Chapter 6.

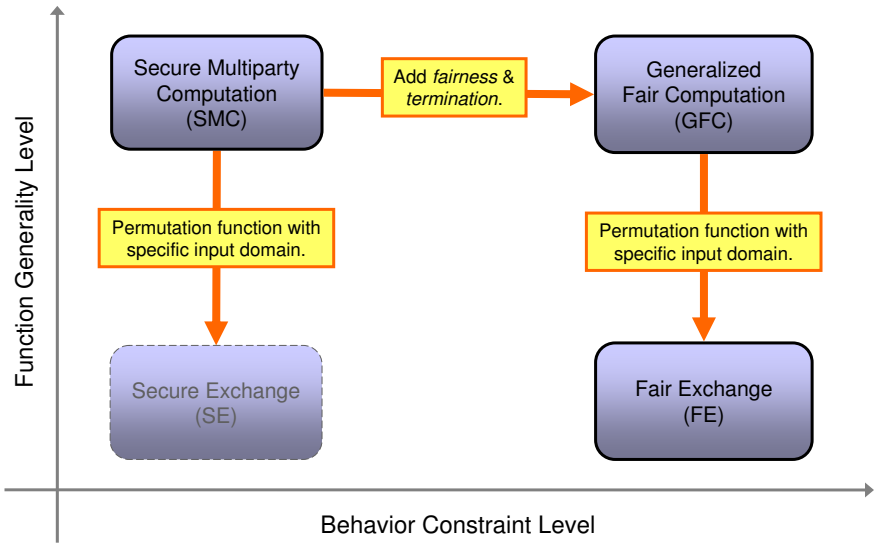


Figure 5.3: Illustration of the generality and security levels of the problems and their relations.

As illustrated in Figure 5.3, the GFC problem encompasses the same level of abstraction as SMC, while being strictly stronger than SMC. Fair exchange is simply a specific instance of the GFC problem. Similarly, a problem combining the guaranties of SMC and the permutation function of FE would be a specific instance of SMC, named *secure exchange* for example.

5.6 Origin of the Confusion

After describing and comparing secure multiparty computation and fair exchange from the perspective of a common specification framework, we can now revisit the origins of the confusion between them. From their respective literature, it is understandably difficult to distinguish precisely one from the other, since they have much in common but are described using different vocabularies. The main reason is probably to be found in the close connection of both with the notion of *trusted third party* (TTP). In SMC, the TTP is the ideal to reach for, and in FE, it is the only unconditional solution to the problem. Moreover, a TTP ensures a large set of security properties but the focus of each problem is on a different subset of these properties. Hereafter, we propose a discussion, hopefully a clarification, on the possible origins of confusion: (1) the seemingly contradictory results and (2) the TTP emulation approach taken in SMC.

5.6.1 Contradictory Results?

On the one hand, when assuming an honest majority, SMC can be made unconditionally secure, even in the context of *secure computation with unanimous abort and complete fairness* [RB89], i.e., fair computation. On the other hand, as shown in Section 1.4, fair exchange is impossible to solve without at least one identified process that all processes can trust. So either one of these results is wrong or they are simply not addressing the same issue. Indeed, if these two statements seem contradictory, it is because of their common context and the lack of accuracy found in natural language, i.e., the notion implied by the word *fairness* is different from one case to the other.

In the case of fair exchange, in order to respect the terms of the exchange, the notion of fairness implies a constraint on the *inputs* of all processes, including those of Byzantine processes. There is no such constraint in the case of *fair computation*, since the validity of inputs of Byzantine processes is usually not much of a concern. The concern is to ensure that adversaries may not obtain information from the inputs of correct processes, i.e., either in absolute (*privacy*) or with respect to the information correct processes are able to obtain

(*fairness*). In other words, what makes fair exchange more difficult to solve than fair computation is the constraint on the inputs of all processes. The verification of all the inputs is a key step in solving fair exchange in Byzantine environments and needs to be done by some trusted process, be it a TTP or trustees. This decisive difference in the notion of *fairness* also explains why the domain of the function describing the SMC problem is usually not explicitly defined, whereas it is a crucial element in fair exchange.

5.6.2 An Ideal Model?

The simulation approach was introduced in SMC in order to simplify the description of the behavioral constraints. The aim is to provide the same level of security as a TTP but without a TTP, i.e., in a fully decentralized setting. However, results from both fair exchange and secure multiparty computation have shown that achieving the same level of security as a TTP is simply impossible. In order to make the emulation possible, certain properties ensured by the TTP have to be dropped. For example, in the ideal model, malicious processes are assumed to be able to interrupt the computation of the TTP, e.g., by cutting off all the communications with the TTP. While this assumption is made because ensuring that the computation is not interrupted is indeed impossible in the real model, it is nonetheless contradictory with the very notion of TTP.

In the light of the above restrictions on this alleged ideal model, one can argue that introducing such an approach in order to avoid having to produce an exhaustive list of properties *ensured* by SMC, e.g., *correctness* and *privacy*, is slightly less convincing. First because the impossibility of unconditionally emulating a TTP forces us to provide an exhaustive list of properties that should be *ignored* in the TTP, e.g., *fairness* or *termination*. And secondly because it is somewhat misleading to readers, who are not necessarily aware of the implicit *not-so-ideal* model hidden behind the *ideal* model. Furthermore, since the ability to interrupt the TTP in the ideal model is not so important when considering *privacy* or *correctness*, this restriction is not always explicitly stated in SMC papers.

Chapter 6

Generalized Fair Computation

All generalizations are false, including this one.

Mark Twain

Abstract. Here, we present a new problem, named *generalized fair computation*. The specification of this problem results from the fusion of fair exchange and secure multiparty computation presented in the previous chapter, i.e., it bears the generality of secure multiparty computation and the behavioral constraints of fair exchange. Using our integrated specification framework, we describe this problem and show that it can indeed be used as a black box in order to provide a solution for both fair exchange and secure multiparty computation. Finally, we propose a solution to the problem of generalized fair computation based on a secure multiparty computation module and secure boxes.

6.1 Introduction

In order to bridge the conceptual gap between *secure multiparty computation* (SMC) and *fair exchange* (FE), we introduce the idea of *generalized fair computation* (GFC). This arises from the fusion of FE and SMC by combining the functional generality of SMC with the guarantees of *fairness* and *termination* of FE, as illustrated by the Yin-Yang symbol in Figure 6.1.

The goal of the GFC problem is to provide a generalization of the problem of fair exchange. Figure. 6.2 shows a projection of the SMC, GFC, FE problems with respect to two axes: the x-axis represents the level of security, or behavioral constraint, required by the problems, whereas the y-axis captures the

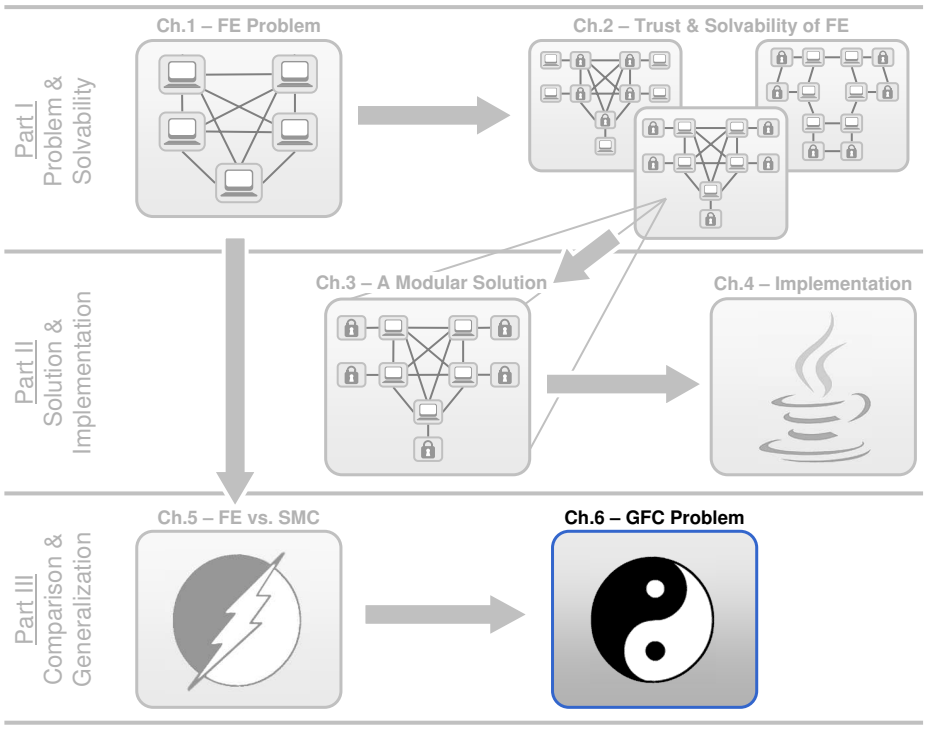


Figure 6.1: Thesis outline.

level of generality of the functional definition of the problems. Accordingly, SMC and GFC are on the same level of generality, while FE is simply a specific instance of GFC that provides the same behavioral constraints. The two other problems represented in Figure 6.2 are *secure exchange* (SE), which was briefly mentioned in the previous chapter, and *fair computation* (FC). While the former was only introduced for illustration purposes, the latter is an important problem of the SMC-related literature and corresponds to *secure computation with unanimous abort and complete fairness* in the hierarchy of [GL02] presented in Section 5.1.

As with GFC and FE, fair computation also provides *termination* and *fairness*, at least to a certain degree. However, while GFC is clearly more general than FE, the difference between fair computation and generalized fair computation may seem unclear. Nonetheless, fair computation, as described in the literature, relates to a subset of the functions captured in GFC, which in particular does not include the permutation function of fair exchange.

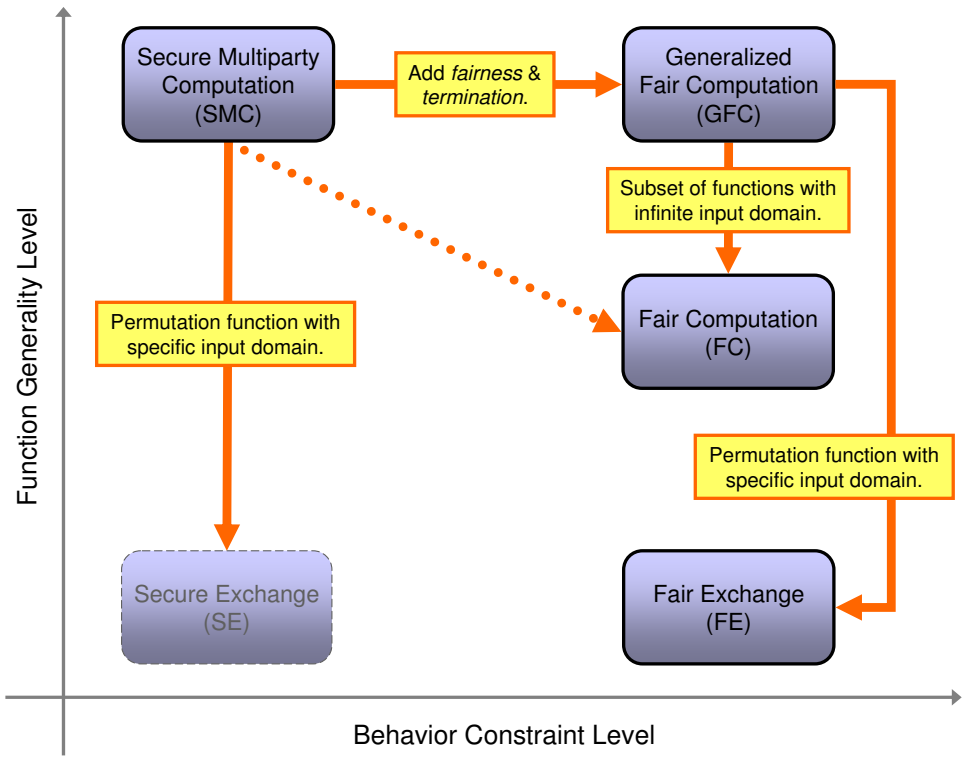


Figure 6.2: The generality and security levels of the various problems.

6.1.1 Fair Computation, no Support for Fair Exchange

In the SMC-related literature, the fair computation problem is indeed obtained by adding *termination* and *fairness* to secure multiparty computation, which in appearance is the same as we suggest doing in order to produce generalized fair computation. As discussed in Section 5.6, the apparent similarity depends on the definition of *fairness*. In the case of fair computation, the weaker definition of fairness actually implies a limited class of functions, i.e., functions that may be computed without the inputs of Byzantine processes [RB89]. Accordingly, including a verification of the input values in the definition of fairness is not necessary. While this particular definition of fairness is sufficient to allow certain specific functions to be computed in a truly fair manner and without relying on a trusted process, it rules out fair exchange as being part of fair computation.

The specification of the fair computation problem can also be defined from the perspective of the GFC problem, i.e., as capturing a subset of the functions of GFC. In this case, the fair computation provides the same properties as GFC, including *fairness*, but the functional definition must explicitly restrict

certain functions in order to match the SMC-related definition of fair computation and its impossibility results and lower bounds. Indeed, in the commonly found definition of fair computation, when assuming an honest majority, the solvability of the problem does not require a trusted process. The problem can even be made unconditionally secure with respect to the number of Byzantine processes by assuming that Byzantine processes have limited computational power compared to correct processes [GMY04].

Interestingly, fair computation can be defined in various ways by changing the level of security in the specification. However, there is a tradeoff between the security level and the generality of the functional definition. In Figure. 6.2, the dotted line between secure multiparty computation and fair computation illustrates the fact that their connection is somewhat unclear. In any case, we argue that the fair computation specification includes an implicit restriction on the function considered, so that GFC is closer to SMC with respect to the generality level of the functional definition, than is fair computation. Moreover, GFC includes the problem of fair exchange, whereas the commonly found definition of fair computation does not.

6.2 The Generalized Fair Computation (GFC) Problem

The generalized fair computation problem consists in a group of processes trying to compute the result of specific function in a fair manner. Intuitively, fair means that, if one process obtains its desired output, then all processes involved in the computation should also obtain their desired output. Similarly to SMC, *privacy* and *correctness* need to be achieved but the main difficulty of GFC resides in achieving *fairness*. The GFC problem also ensures *termination*, since it is necessary to provide true *fairness*, whereas SMC ensures neither.

The specification of GFC is expressed using the integrated specification framework presented in Section 5.2. We thus assume the same system model as in Chapter 5, i.e., a distributed system consisting of a set Π of n fully interconnected processes, $\Pi = \{p_1, \ldots, p_n\}$. The system is *synchronous* and processes may exhibit Byzantine behaviors.

6.2.1 Functional Definition

More formally, each process is required to compute the result of a deterministic function F: $(y_1, \ldots, y_n) = F(x_1, \ldots, x_n)$, where x_i and y_i are respectively the inputs and outputs of process p_i. When all processes are correct, the outcome

of F is defined by a function f with input domain $X = X_1 \times \ldots \times X_n$ and output domain $Y = Y_1 \times \ldots \times Y_n$: $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$. Function f thus provides the outputs of F when the inputs are those expected, i.e., they belong to the domain of f, and when the computation achieves completion.[1] However, if such is not the case, F outputs a special value φ for all processes. This special value φ indicates that the computation aborted. Function F is thus defined as follows:

$$F(x_1, \ldots, x_n) = \begin{cases} f(x_1, \ldots, x_n), & \text{if } \forall x_i,\ x_i \in X_i \text{ and } f \text{ achieves completion;} \\ \forall y_i,\ y_i = \varphi, & \text{otherwise.} \end{cases}$$

6.2.2 Behavioral Constraints

Generalized fair computation allows processes to compute, fairly and securely, the result of a specific function, with each process providing an input value and expecting to receive an output value as the result of the computation. The computation is completed when every process outputs either a value y_i, which is by definition the result of function f, or the abort value φ, meaning that the computation aborted. The behavioral constraints of the problem of generalized fair computation are described by the following set of properties.

Validity. If any correct process p_i outputs a value y_i, then either $y_i \in Y_i$ or $y_i = \varphi$.

Uniqueness. No correct process outputs more than one value.

Non-triviality. If all processes are correct, no process outputs the abort value φ.

Termination. Every correct process *eventually* outputs a value.

Privacy. No process p_j outputs the input value x_i or the output value y_i of any correct process p_i, apart from what is possibly given away by inputs and outputs of Byzantine processes.

Fairness. If any process p_i outputs a value y_i, with $y_i \in Y_i$, then every correct process p_j outputs a value y_j, with $y_j \in Y_j$, unless p_i is Byzantine and y_i is computable from the inputs of Byzantine processes.

The addition of the *termination* and *fairness* properties to the list of properties of SMC has an influence on the *validity* property. Indeed, *termination*

[1] As for FE, certain Byzantine behaviors may interfere with the computation of f, possibly preventing its completion.

implies having a special abort value in cases where executions do not proceed as expected. This obviously impacts the *validity* property which needs to be modified to take into account the special outcomes.

Furthermore, the functional definition also has an impact on the *validity* property, since it requires that all inputs be verified. Accordingly, in the *validity* property, the correct outcome y_i is computed with the inputs of all processes, not just with at least the inputs of correct ones. However, in the case of certain functions that may be correctly computed in the absence of inputs from Byzantine processes, the input verification would arbitrarily restrict the correct outcomes. A simple countermeasure to this drawback is achieved by introducing an omission value $\perp$ in each element X_i of the domain X. The value $\perp$ is then used in the computation of f in place of missing inputs of Byzantine processes.

6.3 Using GFC for Solving SMC and FE

By construction, GFC is a generalization of FE and they both share the same set of properties. Thus, building FE on top of GFC simply consists in defining function f as a permutation, with no fixed points, and in limiting the input and output domains, X and Y, to n singletons matching the terms of the exchange. More interestingly, a solution to SMC can be built on top of GFC, showing that the problem of GFC is indeed stronger than the the problem of SMC.

6.3.1 An Algorithm for SMC based on GFC

Algorithm 6.1 provides a solution to SMC, with function f, relying on a specific GFC module with function $f' = f$. The algorithm is fairly straightforward as it first calls the input primitive of the GFC module and then, if that value is not the abort value φ, it outputs the value provided by the GFC module.

Algorithm 6.1 SMC protocol executed by process p_i

```
1: Uses: Generalized Fair Computation (GFC)
2: function input(x_i)
3:     GFC.input(x_i)                          {call the input primitive of GFC}

4: upon GFC.output(y'_i) do                    {GFC callback}
5:     if y'_i ≠ φ then                        {check if value is not the abort value φ}
6:         output(y'_i)                        {output the result of GFC}
```

Although the GFC module ensures *termination*, Algorithm 6.1 is consistent with the specification of SMC and does not provide *termination*. Indeed, if the output of GFC is the abort value φ, Algorithm 6.1 does not return any value.

6.3.2 Correctness Proof

The correctness proof of Algorithm 6.1 is also fairly obvious since GFC ensures exactly the same *uniqueness*, *non-triviality* and *privacy* properties as SMC. Thus it only remains to show that Algorithm 6.1 ensures the *validity* property of SMC.

Theorem 18 (Validity)**.** *If a correct process p_i outputs a value y_i, then y_i was computed using function f and with at least the inputs of all correct processes.*

Proof. If the SMC module of a correct process p_i outputs a value y_i at line 6, y_i is any value y'_i output by the GFC module at line 4, excluding the abort value φ. From the validity property of GFC, either $y'_i \in Y'$ or $y'_i = \varphi$. From the definition of GFC, when $y'_i \neq \varphi$, y'_i is computed using f' and with the inputs of all processes. So, since $y_i \neq \varphi$ and $f' = f$, y_i was computed using function f and with at least the inputs of all correct processes. □

6.4 Solving GFC in the Absence of a TTP

In order to provide a solution to GFC without a TTP, we propose to complete the model briefly introduced in Section 6.2 with a set Π' of n trustees, $\Pi' = \{p'_1, \ldots, p'_n\}$, which are processes known to be correct a priori. This enhanced system model corresponds to the one detailed in Section 2.2. The need for trustees derives from the difficulty of ensuring *fairness*, discussed in Chapter 2. We showed that the problem of fair exchange is impossible to solve without at least one trusted process, i.e., a correct process identified a priori. While this impossibility result was proven in the context of fair exchange, it is consistent in the context of the GFC problem, since GFC is a generalization of fair exchange. Accordingly, and to depart from the solution based on a centralized trusted third party, we split the necessary trust among processes by introducing the aforementioned set of trustees, where each trustee p'_i is connected to a distinct process p_i, as illustrated in Figure 6.3. In this particular solution, the trustees are merely tamperproof modules hosted by each process, as first introduced in [AV03].

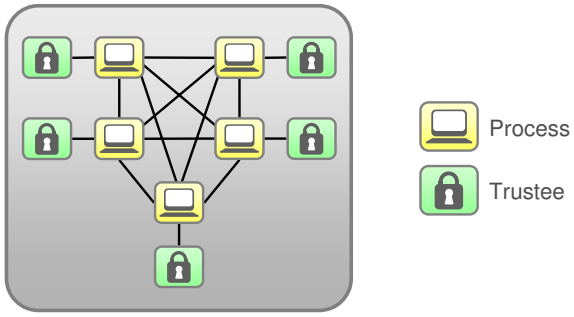


Figure 6.3: Topology with five participants and their trustees.

Communication channels are said to be *perfect links* (PL), which provide *send* and *deliver* primitives (respectively PL.send() and PL.deliver()) and ensure the *reliable delivery*, *no duplication* and *no creation* properties described in Section 1.2. The *synchronous* system assumption implies that delivery will occur within some known time bound Δ_{PL}.

Another important result from Chapter 2 is that the solvability of fair exchange, and by extension GFC, is conditioned by the number of Byzantine processes that a certain topology may sustain. The maximum number b of Byzantine processes sustainable in this particular topology is $b = \lceil \frac{n}{2} - 1 \rceil$, i.e., there must be an *honest majority*.

6.4.1 Relying on Specific Modules

Our GFC protocol relies on three building blocks: an SMC module, as presented in Section 5.3, a Byzantine agreement module and a secure box module, playing the role of the trustee.[2] Figure 6.4 presents the layering of the modules, where only the code of the secure box is tamperproof.

Secure Box (SB)

The secure box module corresponds to the module introduced in Section 3.3. It is a simple tamperproof device corresponding to the notion of trustee. Secure box modules rely completely on their hosts in order to communicate with each other. Thus, a module may be isolated from others, if it is hosted by a Byzantine process. A participant communicates with it by directly invoking primitives. The secure box can be seen as a local service available to each participant. Through the use of a public key infrastructure (PKI), its role is merely one of a safe. The SB device offers the following set of primitives.

[2] In the following, the terms *secure box* and *trustee* are used indifferently.

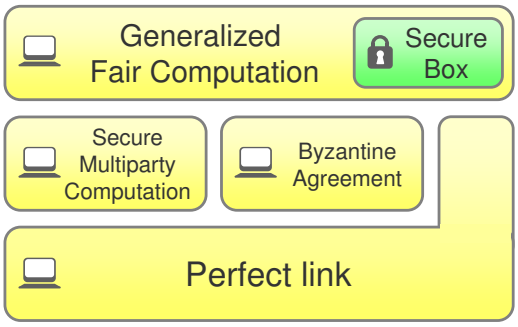


Figure 6.4: A layered diagram of the modules involved in Algorithm 6.2.

SB.isValidOutput(y_i) – Returns a boolean value stating whether the encrypted value y_i is valid, i.e., if $y_j \in Y_i$.[3]

SB.unseal(y_i,proof) – Returns the deciphered value y_j, if the proof of fairness, i.e., a majority of clues issued by other processes, is valid.

SB.sign(m) – Returns a signed version of a message m. Signature is done using the private key of p'_i, i.e., the secure box of process p_i.

Byzantine Agreement (BA)

The Byzantine agreement module is identical to the one presented in Section 3.3. Intuitively, it allows processes to broadcast messages reliably, in spite of Byzantine failures. We briefly recall here the primitives and properties provided by such a module; see the aforementioned section for more details.

BA.start(p_j) – Enables a process p_i to start an execution of BA in order to receive a message from a process p_j. For each execution of the protocol, every correct process calls the *start* primitive at the same time (see timing assumption discussion in Section 2.4) and process p_j then calls the *send* primitive.

BA.send(p_i,m) – Enables a process p_i to broadcast a message m reliably to all processes.

BA.deliver(p_j,S) – Works as a callback and enables a process p_j to receive a set S of messages as the result of a reliable broadcast. Possible outcomes are: (1) S is a singleton, if the sender behaved correctly; (2) S contains

[3]The encryption is done using the module's public key, so the item remains out of reach of the process.

more than one message, if the sender behaved incorrectly by sending different messages to different processes; (3) S is the empty set, if the sender did not send anything.

The goal is to prevent Byzantine processes from threatening agreement among correct processes and ensure the two following interactive consistency (IC) properties.

IC1. If a correct process delivers a set S of messages, then every correct process delivers S.

IC2. If a correct process sends a message m, then every correct process *eventually* delivers the set $\{m\}$.

Secure Multiparty Computation (SMC)

The SMC module provides an implementation of the specification of secure multiparty computation defined in Section 5.3, with function f'. The function is dependent of the desired functional definition of the GFC implementation provided by Algorithm 6.2. Thus, assuming that the functional definition of the GFC problem is function f, function f' is defined by

$$\begin{aligned}(y'_1, \ldots, y'_n) &= f'(x_1, \ldots, x_n) \\ &= (\mathsf{encrypt}_1(f_1(x_1, \ldots, x_n)), \ldots, \mathsf{encrypt}_n(f_n(x_1, \ldots, x_n))).\end{aligned}$$

Intuitively, function f' is identical to function f but the outputs are encrypted using the public keys of the respective secure boxes, i.e., the output of process p_i is encrypted with the public key of its secure box p'_i.

6.4.2 An Algorithm Solving GFC

For sake of simplicity, we assume that all correct processes have local clocks which are synchronized within some fixed maximum drift [PSL80], so they are able to start the algorithm roughly at the same time. We also assume that *upon* actions are executed atomically with respect to one another. Our algorithm is divided into three phases: (1) SMC, (2) voting and (3) clue exchange. The three-phase structure of Algorithm 6.2 is similar to that of Algorithm 3.1. However, the first phase, SMC, replaces the item exchange phase of Algorithm 3.1 and allows for the generalization of the protocol. Thus, an important implication is the replacement at line 14 of the delay Δ_{PL} by the value Δ_{SMC}. Since the system is synchronous, the Δ_{SMC} delay of the first phase is computable as a function of the Δ_{PL} perfect-link delay and function f'.

Algorithm 6.2 Generalized fair computation protocol executed by p_i

```
1:  Uses: Perfect Link (PL), Byzantine Agreement (BA), Secure Box (SB), Secure Multiparty
    Computation (SMC)
2:  Initialisation:
3:      t_0 ← time()                                   {set t_0 to starting time}
4:      output ← FALSE                                 {set variable to false}
5:      sealedValue ← ⊥                                {set variable to null}
6:      votes, clues ← ∅                               {set variables to empty set}

7:  function input(x_i)
8:      SMC.input(x_i)                                 {call the SMC module}

9:  upon SMC.output(y'_i) do                           {SMC callback}
10:     sealedValue ← y'_i                             {store received value}
11:     if SB.isValidOutput(sealedValue) then          {asks SB to check value}
12:         vote ← SB.sign('PROCEED')                  {produce PROCEED vote}
13:         BA.send(p_i, vote)                         {send vote}

14: upon time() > t_0 + Δ_SMC do                       {SMC phase is over}
15:     for all p_j ∈ Π do                             {for all processes}
16:         BA.start(p_j)                              {start BA}

17: upon BA.deliver(p_j, vote) ∧ (¬output) do          {BA callback and not output}
18:     if validProceedVote(vote) then                 {check vote}
19:         votes ← votes ∪ vote                       {add p_j's vote to set}
20:         if (|votes| = n) then                      {if all votes are PROCEED}
21:             for all p_k ∈ Π do
22:                 clue ← SB.sign(votes)              {produce clue}
23:                 PL.send(p_i, p_k, 'CLUE', clue)    {send clue}
24:     else
25:         output ← TRUE                              {set variable to true}
26:         output(φ)                                  {output φ}

27: upon PL.deliver(p_j, p_i, 'CLUE', clue) ∧ (¬output) do   {PL callback and not output}
28:     if validClue(clue) then                        {check if message is valid}
29:         clues ← clues ∪ {clue}                     {add p_j's clue to set}
30:         if (|clues| > n/2) then                    {check if majority of clues}
31:             output ← TRUE                          {set variable to true}
32:             y_i ← SB.unseal(sealedValue, clues)    {unseal value using SB}
33:             output(y_i)                            {output the unsealed value}
```

SMC phase. The first phase of the algorithm allows every process to obtain an encrypted version of the output using the SMC module (line 8). The output has to be encrypted, since the receiving process must not be able to have direct access to it upon reception (line 9). The encryption is thus made using the public key of the SB module of the receiving process. The secure box acts as a safe, so that the process only has access to the output at the end of the protocol.

Voting phase. In this phase, process p_i sends its vote to every process to inform them that it holds the expected output, and waits to receive the vote of every process. Thus, once p_i receives the encrypted output, it queries its secure module to validate the output from SMC and starts the voting phase. The process signs and broadcasts its PROCEED vote (line 13) using BA, indicating that it has received the expected output. It also starts BA for each process in order to synchronize with all the other correct processes. Upon reception of a vote, the validProceedVote() function checks if the delivered set is a singleton containing the valid PROCEED vote of the sender (line 18). If the vote is valid, it is added to the set of votes. Once all the votes are gathered, a process knows that every process has voted PROCEED and has thus received the correct output. With that information, process p_i signs and sends the n votes – called the i-th clue – to every process (line 23). This is necessary in order to enter the final phase, which consists in having processes exchange their clues. Note that nothing prevents some Byzantine process p_k from producing its vote and its clue without having previously received its output. However such behavior cannot prejudice any process other than p_k.

Clue exchange phase. In this phase, process p_i sends its clue to every process and waits to receive the clues from a majority of processes (line 30). Upon reception of a clue, the validClue() function checks if the clue contains a signed set of all n PROCEED votes (line 28). With $\lceil \frac{n+1}{2} \rceil$ clues, it asks its secure module to release the sealed output by deciphering it using its private key (line 32). The majority is necessary to ensure that at least one correct process was able to produce its i-th clue in order for any process to release its item. At this stage, correct processes should be able to release without the help of any Byzantine process; on the contrary, Byzantine processes should not be able to release without the help of at least one correct process.

6.4.3 Correctness Proof

In the following, we prove that Algorithm 6.2 solves GFC in the presence of b Byzantine processes, with $b < \frac{n}{2}$. The restriction on the number of Byzantine processes is called the *honest majority* assumption. Our correctness proof aims at showing that Algorithm 6.2 preserves the *validity*, *uniqueness*, *non-triviality*, *termination*, *privacy* and *fairness* properties of GFC. Based on Lemma 5, the respective theorems hereafter validate each of these properties. The notation p'_i describing a trustee or a secure box is equivalent to the SB notation used in Algorithm 6.2.

Lemma 5. *If some correct process p_j does not receive an encrypted value y'_j at line 9, with $y'_j \in Y_j$, then no process releases at line 33 of Algorithm 6.2.*

Proof. If some correct process p_j does not receive an encrypted value y'_j at line 9, with $y'_j \in Y_j$, it does not send a PROCEED vote. Thus no process obtains the PROCEED vote from p_j. Since no process receives all n PROCEED votes, and since a clue is composed of a signed set of n PROCEED votes, no process produces its clue. From the *no creation* property of perfect links, no process receives any clue. Without a majority of clues, no process is able to unseal the item at line 32 and hence to release the item at line 33 of Algorithm 6.2. □

Theorem 19 (Validity)**.** *If any correct process p_i outputs a value y_i, then either $y_i \in Y_i$ or y_i is the abort item φ.*

Proof. A correct process p_i explicitly outputs the abort item φ at line 26, and the only other case of output is at line 33. In the latter case, p_i outputs the value that is stored in variable sealedValue. From Lemma 5, if p_i outputs at line 33, it has previously received a value y'_i, with $y'_i \in Y_i$, and stored it in variable sealedValue at line 10. Since, from the *uniqueness* property of SMC, the callback from SMC is triggered only once at line 9, p_i outputs a value y'_i, with $y'_i \in Y_i$. □

Theorem 20 (Uniqueness)**.** *No correct process outputs more than one value.*

Proof. The boolean variable output and the atomic execution assumption prevent any correct process from outputting more than one value. □

Theorem 21 (Non-triviality)**.** *If all processes are correct, no process outputs the abort value φ.*

Proof. Since all processes are correct, each process inputs a value x_i at line 8, with $x_i \in X_i$. From the *non-triviality* property of SMC and the synchronous system assumption, every process p_j receives an encrypted value y'_j, with $y'_j \in Y_j$, before time $t_1 = t_0 + \Delta_{\text{SMC}}$,[4] so every process produces and sends in a timely fashion its PROCEED vote at line 13. From the IC2 property of BA, no process receives an invalid PROCEED vote. Therefore, no process outputs the abort value φ (line 26). □

Theorem 22 (Termination)**.** *Every correct process* eventually *outputs a value.*

Proof. The time-out at line 14 ensures that every correct process starts all n executions of BA at the same time. This implies that, from the existence of a time bound for the termination of BA and the IC1 property of BA, there is a time after which, either every correct process receives at least one invalid

[4]Since the system is synchronous, a reasonable delay Δ_{SMC} is computable as a function of the perfect link delay Δ_{PL} and function f'.

PROCEED vote and releases the abort value φ, or every correct process receives all n valid PROCEED votes. In the latter case, every correct process then produces and sends its i-th clue at line 23. From the *reliable delivery* property of perfect links and the *honest majority* assumption, every correct process receives a majority of clues and then outputs at line 33. □

Theorem 23 (Privacy)**.** *No process p_j outputs the input value x_i or the output value y_i of any correct process p_i, apart from what is possibly given away by inputs and outputs of Byzantine processes.*

Proof. Only during the execution of the SMC module, i.e., at line 8, are the input and output values of a correct process p_i (possibly) transmitted through the network. From the *privacy* property of SMC, during that execution, no process p_j obtains the input value x_i or the output value y_i of any correct process p_i, apart from what is possibly given away by inputs and outputs of Byzantine processes. Hence nor does p_j output such values. □

Theorem 24 (Fairness)**.** *If any process p_i outputs a value y_i, with $y_i \in Y_i$, then every correct process p_j outputs a value y_j, with $y_j \in Y_j$, unless p_i is Byzantine and y_i is computable from the inputs of Byzantine processes.*

Proof. The proof is by contradiction.
Assume that some correct process p_j does not output a value y_j, with $y_j \in Y_j$, and that some other process p_i outputs a value y_i, with $y_i \in Y_i$ and y_i not computable from the inputs of Byzantine processes. From assumptions on SB and because y_i is not computable from the inputs of Byzantine processes, p_i needs a majority of clues in order to output y_i. Thus, if p_i outputs y_i (line 33), p_i receives a majority of clues in some previous steps. From the *honest majority* assumption, at least one of these clues is produced by some correct process p_x. Process p_x therefore received all n PROCEED votes. Thus, from the IC1 property of BA, every correct process receives all n PROCEED votes. This implies that no correct process outputs the abort value φ (line 26), including p_j. From the *validity* and *termination* properties of GFC, if p_j does not output the abort value φ, then p_j outputs a value y_j, with $y_j \in Y_j$. A contradiction. □

Conclusion

I think it's the excitement only a free man can feel, a free man at
the start of a long journey whose conclusion is uncertain.
I hope I can make it across the border.
I hope to see my friend, and shake his hand.
I hope the Pacific is as blue as it has been in my dreams.
I hope.

Morgan Freeman as Red, *"The Shawshank Redemption"*

Research Assessment

The aim of the thesis was to provide a study on the solvability of the problem of fair exchange in the context of a synchronous system model with Byzantine failures.

Analyzing the Solvability of Fair Exchange

The first contribution of the thesis was to present a study of the solvability of fair exchange in a general context and its links with the notion of trust. Discussion of solutions and solvability conditions requires a clearly defined problem. As argued, definitions found in the literature were somewhat inadequate for our use, since they were not relevant for any number of processes. We thus introduced a novel specification of the problem of fair exchange composed of elemental properties.

The first step towards discussing the solvability of fair exchange was to show that it is simply impossible in a context where no process can be trusted. This impossibility result was obtained despite assuming a fully connected network, lessening the potential power of Byzantine processes over the communication channels. The two solutions – the *trusted third party* (TTP) [BP90] and the

guardian angels [AV03] – provided the basis for our extension of the general model. The usual distributed system model, composed of a set of processes, was completed with a set of trusted processes (trustees), known to be correct a priori. This allowed a flexible representation of the notion of trust, which we showed was necessary in order to solve fair exchange. Furthermore, the network topology assumptions were generalized to connected graphs, allowing for a greater diversity of settings.

In this new context, we defined the *reachable majority* condition, a connectivity condition stating that, for any failure pattern, each correct processes must be connected by a reliable path to a majority of trustees. We then showed that fair exchange is solvable *if and only if* the reachable majority condition is met. In order to prove the second part, sufficiency, we provided a general solution achieving fair exchange for any network topology and any maximum number of Byzantine processes respecting the reachable majority condition. Solvability of fair exchange is thus linked to the tradeoff between the connectivity of a given network topology and the number of Byzantine processes.

Providing a Realistic Solution

With a better understanding of the conditions under which fair exchange is solvable, our next step was to provide a fully-decentralized, yet realistic, solution. This required a specific network topology, similar to the one assumed in the *guardian angels* solution [AV03]. The context was thus of a fully connected network of processes, with each trustee connected exclusively to its corresponding process. The idea behind this setting was to decentralize the trust among processes fully by having each of them host an embedded tamperproof module playing the role of the trustee.

The second step towards envisaging real deployment of the solution through, for example, the use of smart cards, was to limit the role of the trustee as much as possible. We thus presented an algorithm solving fair exchange that only relies on the tamperproof modules in limited key steps of the protocol. This modular decentralized algorithm can be seen as a building block and a first step towards achieving peer-to-peer m-business solutions.

An implementation was then provided in the context of a pedagogical visualization tool, in which executions of our fair exchange protocol can be configured and monitored through a graphical display. While the application illustrates the difficulty of fair exchange, its realization also allows for the description of a procedure to implementing a wide range of Byzantine behaviors.

Discussing Related Problems

While we were confident in our results on the solvability of fair exchange, results in the field of modern cryptography for the problem of secure multiparty computation were found to be strangely contradictory to ours, since both problems are at first glance quite similar. This forced us to go back and investigate the specifications of the problems.

The main difficulty in comparing fair exchange and secure multiparty computation is that the descriptions and approaches used in their respective fields differ greatly. We thus introduced a common specification framework in order to describe both problems, which allowed us to distinguish better one from the other and threw light on the possible origins of the confusion between them.

The comparison of fair exchange and secure multiparty computation led us to describe the problem of generalized fair computation, a generalization of fair exchange. We solved the new problem by generalizing our solution to fair exchange with a module implementing secure multiparty computation.

Future Research

Asynchronous fair exchange? We have seen that fair exchange is a difficult problem that was shown to be impossible in certain contexts. While assuming a *synchronous* system model is not sufficient to allow fair exchanges in the absence of trust, it plays an important role for ensuring *termination*, a crucial property for providing *fairness*. However, possible future work would investigate fair exchange in the context of an *asynchronous* system model with failure detectors. As shown in [DGG02], failure detection in the context of Byzantine failure is problematic, nonetheless there is some room for partial Byzantine failure detection.

Extending the solution to GFC? Another possible direction for future research is to provide a solution to generalized fair computation in the context of a model assuming connected graphs, i.e., extended from the *fully connected* case. If we consider the two fair exchange protocols of Chapters 2 and 3, and the GFC protocol of Chapter 6, we see that the general solution of Chapter 2 and the GFC solution can be viewed as generalizations of the modular solution of Chapter 3. The modular algorithm is a solution to fair exchange in the context of a specific network topology, where processes are fully connected and trustees are only connected to their corresponding processes. The general solution to fair exchange of Chapter 2 is thus a generalization with respect to assumptions on the network topology, whereas the GFC solution is a

generalization of the functional definition. Combining the two generalizations might thus produce a solution to GFC for any network topology that can be represented as a connected graph.

Graceful degradation? When producing our modular solution to fair exchange, the focus was on trying to reduce as much as possible the use of the secure box modules. However, the specific network topology implies that the solution requires an honest majority of processes. If a half or more processes are Byzantine, fairness is simply no longer ensured. As we have shown this is indeed a lower bound in order to ensure true fairness, nonetheless an interesting feature of the *guardian angels* protocol is its ability to degrade gracefully [AGGV05]. A possible extension to our work would thus be to implement the graceful degradation of [AGGV05], while preserving the limited role of the secure boxes.

Final Word

The aim of the thesis was indeed to provide a study on the solvability of the problem of fair exchange in the context of a synchronous system model with Byzantine failures. This led us on the long winding path of research, with its aches and pains, its trials and errors, its ups and downs. While, on the verge of reaching the end, the destination is now felt to be a worthwhile achievement, it can only give the slightest glimpse of what has been discovered during the journey. However, for the last step, we have concluded with the concrete – our research contributions – since experience and personal growth can only be the beginning of the next journey.

Bibliography

[ACT98] M. K. Aguilera, W. Chen, and S. Toueg. *Failure detection and consensus in the crash-recovery model.* In International Symposium on Distributed Computing, pages 231–245, 1998.

[AGG+04] G. Avoine, F. Gärtner, R. Guerraoui, K. Kursawe, S. Vaudenay, and M. Vukolic. *Reducing fair exchange to atomic commit.* Technical report, Swiss Federal Institute of Technology (EPFL), 2004.

[AGGV05] G. Avoine, F. Gärtner, R. Guerraoui, and M. Vukolic. *Gracefully degrading fair exchange with security modules (extended abstract).* In Proceedings of the 5th European Dependable Computing Conference - EDCC 2005, 2005.

[Alh98] S. S. Alhir. *UML in a nutshell: a desktop quick reference.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.

[ASW00] N. Asokan, V. Shoup, and M. Waidner. *Optimistic fair exchange of digital signatures.* IEEE Journal on Selected Area in Communications, 18:593–610, 2000.

[Ate99] G. Ateniese. *Efficient verifiable encryption (and fair exchange) of digital signatures.* In CCS '99: Proceedings of the 6th ACM conference on Computer and communications security, pages 138–146, New York, NY, USA, 1999. ACM Press.

[AV03] G. Avoine and S. Vaudenay. *Fair exchange with guardian angels.* In Proceedings of the 4th International Workshop on Information Security Applications (WISA), volume LNCS. Springer, 2003.

[Baj02] S. Bajikar. *Trusted Platform Module (TPM) based Security on Notebook PCs – White Paper.* Intel Corporation – Mobile Platforms Group, 2002.

[BDM98] F. Bao, R. H. Deng, and W. Mao. *Efficient and practical fair exchange protocols with off-line TTP.* In RSP: 19th IEEE Computer Society Symposium on Research in Security and Privacy, 1998.

[BGK04] Z. Benenson, F. Gartner, and D. Kesdogan. *Secure multi-party computation with security modules.* Technical report, RWTH Aachen University of Technology, 2004.

[BGW88] M. Ben-Or, S. Goldwasser, and A. Wigderson. *Completeness theorems for non-cryptographic fault-tolerant distributed computation.* In STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing, pages 1–10, New York, NY, USA, 1988. ACM Press.

[BP90] H. Bürk and A. Pfitzmann. *Value exchange systems enabling security and unobservability.* Computers & Security, 9(9):715–721, 1990.

[Bur92] S. Burbeck. *Applications programming in smalltalk-80: How to use model-view-controller(mvc)*, 1992. http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html, retrieved Mars 23, 2007.

[BWW00] B. Baum-Waidner and M. Waidner. *Round-optimal and abuse free optimistic multi-party contract signing.* In Automata, Languages and Programming, number 1853 in Lecture Notes in Computer Science (LNCS), pages 524–535. Springer, 2000.

[Can00] R. Canetti. *Security and composition of multiparty cryptographic protocols.* Journal of Cryptology: the journal of the International Association for Cryptologic Research, 13(1):143–202, 2000.

[CCD88] D. Chaum, C. Crépeau, and I. Damgard. *Multiparty unconditionally secure protocols.* In STOC '88: Proceedings of the 20th ACM symposium on Theory of computing, pages 11–19, New York, NY, USA, 1988. ACM Press.

[CT96] T. D. Chandra and S. Toueg. *Unreliable failure detectors for reliable distributed systems.* Journal of the ACM, 43(2):225–267, 1996.

[DFS05] V. Drabkin, R. Friedman, and M. Segal. *Efficient byzantine broadcast in wireless ad-hoc networks.* In DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05), pages 160–169, Washington, DC, USA, 2005. IEEE Computer Society.

[DGG02] A. Doudou, B. Garbinato, and R. Guerraoui. *Encapsulating failure detection: From crash to byzantine failures.* In Ada-Europe '02: Proceedings of the 7th Ada-Europe International Conference on Reliable Software Technologies, pages 24–50, London, UK, 2002. Springer-Verlag.

[DGG05] A. Doudou, B. Garbinato, and R. Guerraoui. *Dependable Computing Systems: Paradigms, Performance Issues, and Applications*,

chapter Tolerating Arbitrary Failures with State Machine Replication, pages 27–56. Wiley, 2005.

[DLP+01] J.G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S.W. Smith, and S. Weingart. *Building the IBM 4758 secure coprocessor.* Computer, 34(10):57–66, 2001.

[DR82] D. Dolev and R. Reischuk. *Bounds on information exchange for byzantine agreement.* In Proceedings of the 1st symposium on Principles of distributed computing (PODC'82, pages 132–140, New York, NY, USA, 1982. ACM Press.

[DS83] D. Dolev and H. R. Strong. *Authenticated algorithms for byzantine agreement.* SIAM J. Comput., 12(4):656–666, 1983.

[EY80] S. Even and Y. Yacobi. *Relations among public key signature systems.* Technical report, Technion - Israel Institute of Technology, 1980.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. *Impossibility of Distributed Consensus with One Faulty Process.* J. ACM, 32:374–382, April 1985.

[FM88] P. Feldman and S. Micali. *Optimal algorithms for byzantine agreement.* In STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing, pages 148–161, New York, NY, USA, 1988. ACM Press.

[Fow03] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition.* Addison-Wesley Longman Ltd., 2003.

[FR97] M.K. Franklin and M.K. Reiter. *Fair exchange with a semi-trusted third party (extended abstract).* In CCS '97: Proceedings of the 4th ACM conference on Computer and communications security, pages 1–5, New York, NY, USA, 1997. ACM Press.

[GHM+99] R. Guerraoui, M. Hurfin, A. Mostéfaoui, R. Oliveira, M. Raynal, and A. Schiper. *Consensus in asynchronous distributed systems: A concise guided tour.* In Advances in Distributed Systems, Advanced Distributed Computing: From Algorithms to Systems, pages 33–47, London, UK, 1999. Springer-Verlag.

[GL90] S. Goldwasser and L.A. Levin. *Fair computation of general functions in presence of immoral majority.* In CRYPTO '90: Proceedings of the 10th Annual International Cryptology Conference on Advances in Cryptology, pages 77–93, London, UK, 1990. Springer-Verlag.

[GL02] S. Goldwasser and Y. Lindell. *Secure computation without agreement.* In DISC '02: Proceedings of the 16th International Conference

on Distributed Computing, pages 17–32, London, UK, 2002. Springer-Verlag.

[GMY04] J. A. Garay, P. MacKenzie, and K. Yang. *Efficient and secure multi-party computation with faulty majority and complete fairness*. Cryptology ePrint Archive, Report 2004/009, 2004. http://eprint.iacr.org/.

[Gol04] O. Goldreich. *The Foundations of Cryptography*, volume 2. Cambridge University Press, 2004.

[GR06a] B. Garbinato and I. Rickebusch. *Impossibility results on fair exchange*. In Proceedings of the 6th International Workshop on Innovative Internet Community Systems (I2CS'06), volume LNI. German Society of Informatics, 2006.

[GR06b] B. Garbinato and I. Rickebusch. *Orchestrating fair exchanges between mutually distrustful web services*. In Proceedings of the ACM Workshop on Secure Web Services (SWS'06). ACM Press, 2006.

[GR06c] B. Garbinato and I. Rickebusch. *A topological condition for solving fair exchange in byzantine environments*. In Proceedings of the 8th International Conference on Information and Communications Security (ICICS'06), volume LNCS. Springer, 2006.

[GR06d] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag, Berlin, Germany, 2006.

[GR07] B. Garbinato and I. Rickebusch. *Secure multiparty computation vs. fair exchange: Bridging the gap*. Technical Report DOP-20070123, University of Lausanne, DOP Lab, 2007. http://www.hec.unil.ch/dop/Download/articles/DOP-20070123.pdf.

[HT93] V. Hadzilacos and S. Toueg. *Fault-tolerant broadcasts and related problems*. pages 97–145, 1993.

[KK06] J. Katz and C.-Y. Koo. *On expected constant-round protocols for byzantine agreement*. Cryptology ePrint Archive, Report 2006/065, 2006.

[LLR02] Y. Lindell, A. Lysyanskaya, and T. Rabin. *On the composition of authenticated byzantine agreement*. In STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing, pages 514–523, New York, NY, USA, 2002. ACM Press.

[LSP82] L. Lamport, R. Shostak, and M. Pease. *The byzantine generals problem*. ACM Transactions on Programming Languages and Systems, 4(3):382–401, July 1982.

[MGK02] O. Markowitch, D. Gollmann, and S. Kremer. *On fairness in exchange protocols.* In Proceedings of the 5th International Conference Information Security and Cryptology (ICISC 2002), volume 2587 of Lecture Notes in Computer Science, pages 451–464. Springer, November 2002.

[Mic03] S. Micali. *Simple and fast optimistic protocols for fair electronic exchange.* In PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing, pages 12–19, New York, NY, USA, 2003. ACM Press.

[MR91] S. Micali and P. Rogaway. *Secure computation (abstract).* In CRYPTO, pages 392–404, 1991.

[PG99] H. Pagnia and F. Gärtner. *On the impossibility of fair exchange without a trusted third party.* Technical report, Swiss Federal Institute of Technology (EPFL), 1999.

[PSL80] M. Pease, R. Shostak, and L. Lamport. *Reaching agreement in the presence of faults.* Journal of the ACM, 27(2):228–234, April 1980.

[RB89] T. Rabin and M. Ben-Or. *Verifiable secret sharing and multiparty protocols with honest majority.* In STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of computing, pages 73–85, New York, NY, USA, 1989. ACM Press.

[RR02] I. Ray and I. Ray. *Fair exchange in e-commerce.* SIGecom Exchanges, 3(2), 2002.

[RRN05] I. Ray, I. Ray, and N. Natarajan. *An anonymous and failure resilient fair-exchange e-commerce protocol.* Decision Support Systems, 39(3):267–292, 2005.

[Sha79] A. Shamir. *How to share a secret.* Commun. ACM, 22(11):612–613, 1979.

[SXL05] M. Srivatsa, L. Xiong, and L. Liu. *Exchangeguard: A distributed protocol for electronic fair-exchange.* In 19th International Parallel and Distributed Processing Symposium (IPDPS 2005). IEEE Computer Society, April 2005.

[Yao82] A.C. Yao. *Protocols for secure computation.* In 23th Symposium on Foundations of Computer Science (FOCS), pages 160–164. IEEE Computer Society Press, 1982.